This post describes the process in which validators obtain shared keys to be used in secret contracts. For the time being, we assume all keys are derived from a single source of true randomness (a master seed

). We discuss this more in the end.

From the shared seed, validators can generate additional keys/seeds pseudo-randomly using CSPRNGs/KDFs. More on this later.

A key design choice in this protocol was to remove the need for off-chain communication. Everything needs to be handled on-chain for simplicity and consistency. This may shed some light on why we describe a process that requires up to 3 transactions to register a new validator.

Bootstrap protocol

For now, let's start with the bootstrapping

protocol to get a shared random seed to all validators.

Init (new network, no existing validators)

1. When the first validator joins in, they need to look at the blockchain's state and determine that there are no existing registered validators.
2. In the enclave, call register_validator(first_validator=True). The code does the following steps:
3. sk_v := rand() // generate a 256-bit secret key for this validator
4. pk_v := sk_v*G // generate the corresponding public key
5. seal(sk_v, pk_v) // persist this key pair locally, encrypted using the enclave's hardware key
6. seed := rand() // generate a 256-bit master seed
7. seal(seed) // persists locally
8. return (pk_v, remote_attestation_quote)
9. Outside of the enclave, the validator should send the quote to Intel's Attestation Service (IAS) - potentially through Enigma's proxy as was the case with Discovery, and get a signed report

.

1. Broadcast a register_validator(pk_v, report)

transaction.

1. On-chain: check that the report is valid on-chain. If it is, and because there are no other validators, immediately approve this validator.

New validator joins (at least one validator exists)

Phase 1:

1. Look at the blockchain's validator registry, see that at least one exists.
2. Enclave: call register_validator(first_validator=False)

. Code proceeds as above, but does not create/seal a new seed

1. Same as above.
2. Same as above, but the validator should include some SCRT bounty so other validators are incentivized to proceed with Phase 2. NOTE: we should allow the validator to withdraw their stake after X blocks if no one shared the key with them.
3. On-chain: check that the report is valid on-chain. If it is, and because there ARE other validators, move the new validator to a 'pending confirmation' state.

Phase 2:

1. Any registered validator can see the pending validator's request on chain. Let's say the pending validator is called v2

and the registered validator is called v1

. v1 should then call share_key(pk_v2, report)

,which does the following:

- Checks that v2's report is valid.
- (symm_key, pk) := derive_key(pk_v2) // ECDH key derivation, which creates a new fresh key pair, and uses that and v2's pubkey to generate an symmetric key pair (symm_key

) that can be used to encrypt the shared seed. pk

is the public key of the fresh key pair.

- enc_seed := encrypt(symm_key, seed)
- return (enc_seed, pk, proof) // TODO: explain proof - needs to sign the payload that includes v2's pubkey.
- Broadcast share_seed(enc_seed, pk, proof)

tx to the chain

1. On-chain:
2. Validate the proof came from a proper enclave. Essentially, need to validate that v1 signed the payload, and that the payload includes the call to share_seed

with pk_v2

as an argument

- Store in the state (pk_v2 --> enc_seed) and release the bounty to v1

Phase 3:

1. When v2 sees enc_seed has been committed on-chain by v1, they can call store_seed(enc_seed, pk)

in the enclave, which does the following:

- symm_key := derive_key(sk_v2, pk)
- seed := decrypt(symm_key, enc_seed)
- seal(seed)
- Send a confirm_validator

tx on-chain.

1. On-chain: move the validator from 'pending' status to 'confirmed' status (i.e., finalize registration).

The protocol above essentially allows all validators to share a truly random seed that is only ever accessible from a validator's enclave.

One potential downside is that there are no limitations to registering a validator, and so anyone

can create it and get the network's seed in their enclave. This increases the attack surface, so it may be desirable to add a check on-chain that only validators with a minimum stake

are able to go through this process. This is also closer to the version we had with Discovery, where workers had minimum stake - and the reasoning was also to increase security of the system.

Using this mechanism, our threat model is somewhat better, as there's a barrier for even obtaining the shared key - you have to either be validator with enough stake, or you need to collude with a validator with enough stake (which essentially reduces to the same threat model --> we assume that anyone with stake X+ cannot or will not compromise their enclave).

Pseudo-random keys/seeds

With a shared seed, it's easy to reach consensus implicitly on shared keys that the validators need. For this, we need a cryptographically secure PRNG (CSPRNG). We can either use Rust's native implementation of a CSPRNG. They provide two of those in the Rand library, or we can use a stream/block cipher in the ring

library which we already use. For simplicity, I suggest we use ChaChaRng

.

With the single shared 256-bit seed generated above, we can run the CSPRNG several times to get other 256-bit pseudo-random keys of interest:

1. First 256 bits: used to generate (sk_io, pk_io)

--> a key pair whose pubkey can be used by users to derive new symmetric encryption keys for encrypting input/outputs. The protocol above needs to be changed so that the first validator also broadcasts pk_io to the network

.

1. Next 256 bits: used to generate master_state_key

--> a symmetric key used to encrypt contract state. In practice, this is a seed we can use to derive further keys.

1. Next 256 bits: used to generate master_iv

--> this is a seed that can be used to generate fresh IVs for encrypting outputs. That's how the network can avoid non-determinism and still maintain the security of symmetric ciphers.

1. Next 256 bits: used to generate master_rand_seed

--> this is a seed that can be used to generate randomness. TODO: discuss the problems of doing this in a second post, if we decide to combine it with a KDF (user manipulation issue…).

Better key management policies

In the future, we can potentially increase security by allowing more elaborate key management schemes. The main idea is to better protect the master seed, from which important keys/other seeds are generated. This can be achieved for example by:

1. The seed itself should be handled in a separate enclave and only output fresh keys in an enclave-to-enclave communication. The enclave holding the seed should have as little code as possible, audited against potential side-channel attacks. This is easier if it's limited to holding the master seed and generating seeds/deriving one-time keys.
2. Rotating the seed.
3. Localizing the seeds (different seeds for different quorums - this is closer to how Discovery handled it, but this becomes overly complicated very fast).
4. Using secret sharing/mobile proactive secret sharing.

These are just examples, and they are not fully fleshed out. This is an open area of research and we invite the community to propose ideas/research/help develop this

.