This is a work-in-progress draft by Shunfan Zhou and Hang Yin. We welcome comments and experimentation with the idea.

In [Early Thoughts on Decentralized Root-of-Trust

](https://collective.flashbots.net/t/early-thoughts-on-decentralized-root-of-trust/3868), we described our motivation to update the TEE Root-of-Trust from hardware-kept unrecoverable secret keys to a smart-contract-governed software (i.e., DeRoT). In this article, we'll delve deeper into the details of its key management protocol.

Threat Model

A fundamental assumption in our threat model is that TEE can be compromised

. Therefore, our primary goal is to ensure that even if the TEE hardware is compromised, the application should still function correctly with minimal data leakage. This is why we introduce Decentralized Root-of-Trust (DeRoT), a verifiable Key Management Service (KMS) for TEE governed by smart contracts, to replace the hardware-kept unrecoverable secret keys in TEE.

Based on this assumption, we discuss potential threats and corresponding countermeasures. Specifically, both the confidentiality and integrity guarantees provided by TEE hardware can be compromised.

- If confidentiality is compromised, encrypted memory values may be leaked, and this will further affect the sealed data on persistent storage. The countermeasures aim to:
- Limit the damage scope when a single worker is compromised to only the minimal amount of data it knows.
- Ensure forward and backward secrecy so that historical and future data remains safe.
- Limit the damage scope when a single worker is compromised to only the minimal amount of data it knows.
- Ensure forward and backward secrecy so that historical and future data remains safe.
- If the integrity promise is broken, an adversary can instruct the TEE to produce false results. In this case, all outputs from the compromised TEE should not be trusted or used by other TEE hardware and clients. While detecting TEE vulnerabilities and compromises is beyond the scope of DeRoT, it should be able to revoke authentication and prevent dirty data from being persisted.

Introducing DeRoT

Compared to traditional key management services, DeRoT differs in two aspects:

- Access Control on Clients:

DeRoT verifies its clients to ensure they are genuine TEE hardware running valid applications.

- Verifiable Chain-of-Trust:

The DeRoT program, along with all its generated keys, must be verifiable through blockchain technology. We will explain how to verify the integrity of the DeRoT program and check whether a key is genuinely generated by DeRoT in the following sections.

DeRoT is designed to enhance the functionalities of the current hardware-kept secret key in TEE in two major scenarios: 1) identifying the TEE hardware, and 2) data sealing (i.e., the encryption of persistent storage). Its advantages include:

- Vendor- and Hardware-Agnostic TEE Verification:

DeRoT is responsible for distributing secret keys only to valid TEE hardware and controlling the key lifecycle. Clients can delegate the Remote Attestation (RA) report verification to DeRoT, needing only to verify whether the key held by the TEE is valid. This simplifies client implementation and abstracts away the details of RA reports. Additionally, this approach can be easily extended to support different TEE hardware and get integrated into existing certificate chains.

- Key Rotation in Data Sealing with Enhanced Security Features:

With key rotation, the time window for attackers is significantly limited. Furthermore, this approach ensures forward and backward secrecy for sealed data, enhancing overall security.

- Support for Upgradable and Migratable Applications:

Since the sealed data is now encrypted with keys generated by the standalone DeRoT, storage can be entirely separated from TEE hardware. This allows upgraded applications to decrypt saved states in a controllable way, and enables dynamic

addition or removal of TEE hardware without concerns about data loss. Additionally, this architecture aligns with the existing cloud paradigm, facilitating easier migration.

Unique Challenge: Stealthy Deployer Attack

Unlike in smart contracts that every operation is observable on-chain, the introduce of DeRoT enables a blockchain-TEE hybrid system where the interactions can happen both on-chain and invisibly off-chain. Also, the support for upgradable application code will introduce more risks since the benign code may be changed to dump the encrypted data in an unconscious way.

Considering the following attack from a malicious application deployer:

1. An open-source benign application is deployed to TEE and trusted by its users to process the privacy data;
2. The deployer asks the DeRoT to upgrade the application code. It seems no reason to reject it since he is the owner;
3. The deployer injects malicious logic to dump all the privacy data in the new code, but doesn't announce this to the public. Since this upgrade is between the TEE and DeRoT, it is not observable from the on-chain perspective;
4. The deployer can make another upgrade to remove the malicious logic to erase the evidence.

Actually, similar attacks on the upgradable smart contracts happened before, but the difference is that such attack will not be naturally perceived when the application and the DeRoT are running off-chain. It's worth noting that adding more access control on the application upgrade operation (e.g. enforce a multi-sig smart contract) cannot mitigate this risk.

This attack motivates an important principle in our design: every operation involving key authorization changes must be initiated on-chain to ensure observability

. This will ensure the security equivalence to existing smart contracts.

Design

Verifiable DeRoT and Key Chain

[

image

1248×558 40.8 KB

](https://collective.flashbots.net/uploads/default/original/2X/8/842c2760a67bbd491e14fa00ca8bfa5116ef4c00.png)

To enable the verification on the DeRoT program integrity and correctness, its measurement will be specified in the governance smart contract. Several steps need to be followed to ensure the smart contract's governance over the DeRoT program.

1. The DeRoT code is open-source for anyone to do code and security review to ensure the correctness of key management logic and no backdoors;
2. The executable must be produced with reproducible build so the verifier can know it matches the source code;
3. The digest of valid executable is published through on-chain governance;
4. Multiple DeRoT instances run inside TEE. Here we only rely on TEE to provide the measurement over the executable, and ensure it matches the digest on chain.

Apart from RA report verification, the core logic of key management in DeRoT is quite compact. It just involves two operations:

- the creation of the RootKey;
- the derivation of each application key using RootKey and nonce including deployer identity and application measurement.

Because of the simple logic, it can use MPC/FHE-based threshold key generation and derivation with no concern about the performance overhead.

The public key of RootKey must be published on-chain to enable any party to verify the validity of application keys with it. We will show that such root-key-derivation pattern is important to enable an efficient key rotation in the following section.

Key Derivation

The key management of DeRoT follows the Principle of Least Privilege

that the secrecy known by each entity in the system is strictly limited to what it requires to finish the jobs. We first explain the keys used during program execution, and then discuss the privileges of different entities.

[

image

1584×768 65.4 KB

](https://collective.flashbots.net/uploads/default/original/2X/5/5a4e4f352de8bea46e7e7d6b580ce4b8d1ddb178.png)

DeRoT turns the previous hardware-centric key management to application-centric key management. That is, each application has its own keys to seal the states and encrypt the databases, and they can be distributed to different TEE workers to enable the applications to be migrated between them. There are three entities involved in the key management protocol, and they have different knowledge of the keys:

- DeRoT, which knows the RootKey and is able to generate any key;
- Storage service, which can be the external storage server or service containing the data and states of the application. The storage service holds the StorageKey and can rotate it independently to keep re-encrypting the data. It's worth noting that the DeRoT can rotate it synchronously if they share the same timestamp;
- The TEE worker running the applications. Every key it holds must be distributed from DeRoT.

An important feature here is that every key received by TEE worker is kept in rotation

, so the DeRoT can passively "revoke" the worker's access to certain program states by not to deliver the newly rotated keys. Once DeRoT rejects to deliver the latest keys, the TEE worker will not able to decrypt the local sealed data or read/write to the external storage.

During program execution, the DeRoT keeps delivering the latest AppKeys to functional workers so they can correctly decrypt the program states. Although this makes DeRoT the single point of failure, we want to argue that since the DeRoT only focuses on key management, it is easier to do thorough code review to ensure its security. Further, as discussed above, the DeRoT only listens to the application deployment request from blockchain and is not accessible to others, which greatly reduces the attack interface.

The RootKey is only known by DeRoT and a Key Derivation Function (KDF) is used to generate all the other keys. The AppRootKey above is merely a logical entity and can be skipped in the actual key generate. For example,

SealingKey_{epoch} = KDF(RootKey, (deployer_id, app_hash, nonce, "seal",epoch))

Key Rotation

There are two rotation variables above: the RootKey itself and by changing epoch

during key derivation.

The rotation of the RootKey will immediately changes all the keys in the system, while the interval of updating epoch

can be configured independently for different storage services so they can be rotated in different frequencies based on the requirements.

The update of the applications, which will change the app_hash

above, can be regarded as a special key rotation for the application.

Forward and Backward Secrecy Ensurance in TEE Workers

On one hand, the TEE worker runtime will only keep the latest AppKeys and remove all the used keys from its memory to preserve the forward secrecy; on the other hand, once a worker is considered compromised, the DeRoT will stop delivered the rotated new keys, so the worker will not be able to get further access to the following code and state updates of the program, preserving the backward secrecy.