TL;DR: We propose a secure and practical deterministic key generation scheme and pseudorandom number generator, from which RSA keys can be generated to simplify key backup and retrieval.

This research is a joint effort from Ethereum Fellows:@Mason-Mind @georgesheth @dennis @AshelyYan.

1. Introduction

Traditionally, cryptographic keys are generated using randomness to ensure unpredictability.

In contrast, deterministic key generation refers to the process of generating cryptographic keys in a deterministic manner. In deterministic key generation, a single starting point (seed) is used to derive the keys using some Key Derivation Function (KDF). It brings some convenience in key management, however, also security risks and privacy concerns. The advantages can be concluded as follows:

- Streamlined key management: Users can generate a sequence of keys from the initial seed, eliminating the necessity to store multiple keys individually. Instead, they only need to safeguard the initial seed in order to secure all the derived keys.
- Simplified key backup and retrieval: Users can regenerate all the cryptographic keys from the initial seed, which simplifies the key retrieval.
- Meet the industry demand. Industry developers are always looking for this solution and has huge value to the product delivery.

However, in this function, if the seed is compromised, every derived key becomes vulnerable to potential threats. Therefore, it is important to keep the initial seed secure. Meanwhile, it is essential that the initial seed remains unpredictable to adversaries, for reasons that are self-evident.

The main difficulty in designing a KDF is the initial keying material. When the source keying material is not uniformly random or pseudorandom, additional preprocessing is needed. In blockchain, there are some applications and solutions. Hierarchical Deterministic Bitcoin Wallets provide an interesting way of managing cryptographic keys, which form a tree structure. At the root, there is a randomly generated master seed. Using the deterministic key generation technique outlined in the BIP32 standard, this master seed can produce child keys. Since all the keys are generated deterministically, the same set of keys can be generated from the master seed. Argon2 is another popular approach for key derivation. It is designed to be resistant against brute-force attacks. Argon2 improves security by using a salt and a password as inputs. The salt is a unique random value. The purpose of using salt is to prevent attackers from using precomputed tables (rainbow tables) to apply brute-force attacks on passwords. Since unique salts are used, even if two users have the same password, their hashed values are still different.

In this proposal, we consider a particular problem of deterministic key generation, which generates RSA keys from ECDSA signatures. We assume that each user has an ECDSA private key sk

, and wants to generate an RSA key pair. Instead of generating the RSA key pair separately, we make the RSA key pair a deterministic function of sk

and a fixed message m

. Our goal is to make the RSA keys secure. In particular, we want the probability of RSA key collisions from different users to be negligible. Due to the nature of RSA key generation, it is not enough to generate a single random number and use it directly as the private key. Instead, we need a pseudorandom number generator. Due to this requirement, we cannot use the approaches that we discussed previously.

To solve this problem, we propose a good deterministic key generation scheme and pseudorandom number generator, from which RSA keys can be generated. From a high level of the scheme, we pick the initial seed to be a signature sig(m, sk)

. The hash of the signature is then used as the key of the AES cipher. The algorithm and security analysis can be found as follows.

2. Algorithm

Our goal is to generate RSA keys securely and deterministically from ECDSA signatures. To achieve this goal, we use the standard procedure to generate RSA keys and a pseudorandom number generator prng

, which provides all sources of randomness. The sequence that prng

generates is a deterministic function of the ECDSA private key sk

and the message m

. Therefore, the same user always generates the same RSA key pair. But from the adversary's point of view, the sequence that prng

generates looks random. Hence the RSA keys are random.

At a high level, we start with an ECDSA signature. We do not assume that the signature is pseudorandom. Instead, we assume it to be unforgeable. This implies that the signatures form a sufficiently large space. (Otherwise, the brute-force attack would succeed with non-negligible probability.) We also assume that SHA256 is a random oracle, which roughly means that the result of the hash function looks random to an adversary unless the adversary knows the preimage of the function. But since we already assumed that ECDSA signatures are unforgeable, the adversary knows the preimage of the hash function with negligible probability. We can safely conclude that the pseudorandom of the result of the hash function, which we use as the key to AES block cipher. We then get a good pseudorandom number generator, from which random keys can be generated.

Algorithm 1 describes the details of the algorithm. Given an ECDSA private key sk

and a message m

, first sign m

using sk

to get an ECDSA signature sig

. We then use SHA256 to hash sig

to get a key

, and set seed

to be the hash of key

. A pseudorandom number generator prng_{sig}

can thus be obtained using AES encryption. We define AES(seed, key)

to be the sequence of AES encryption in counter mode: AES_{ENC}(seed, key)

, AES_{ENC}(seed+1, key)

, AES_{ENC}(seed+2, key)

, \ldots

.

Algorithm 1: Deterministic Key Generation Input: a fixed message m, secret key sk from ECDSA Output: RSA key pair Function: detGenKeyPair(m, sk) sig = ECDSA_Sign(m, sk) key = SHA256(sig) seed = SHA256(key) prng_sig = AES(seed, key) RSAKeyPair(prng_sig)

To generate some RSA key pairs, we use the standard RSA key generation procedure (Algorithm 1). The pseudorandom number generator is responsible for providing all the randomness used in Algorithm 2.

Algorithm 2: RSA Key Generation using prng Function: RSAKeyPair(prng_sig) p <- prgn while p is not a prime do p <- prgn q <- prgn while q is not a prime do q <- prgn Compute n = p*q Compute l(n) = (p-1)*(q-1) Pick some e (e and l(n) are relatively prime) Compute d: e * d = 1 mod l(n) Return: e, d, n

In order to show that detGenKeyPair

returns ``good'' RSA key pairs, we compare our algorithm with a popular reference implementation \textit{Forge}

, which is a native implementation of TLS and various other cryptographic tools in JavaScript. \textit{Forge}

implements a function \textit{forge.pki.rsa.generateKeyPair}

, which generates RSA key pairs. From now on, we will refer to \textit{forge.pki.rsa.generateKeyPair}

as \textit{generateKeyPair}

. Our goal is to show that the key pairs, that deGenKeyPair

generates, are as good as the key pairs, which generateKeyPair generates.

More precisely, we want to show that, for any probabilistic polynomial-time adversary \mathcal{A}

, given Oracle access to the deterministic key generation function deGenKeyPair(\cdot, \cdot)

and generateKeyPair

from generateKeyPair

, \mathcal{A}

cannot distinguish between the keys generated by the two oracles.

3. Security Analysis

3.1 Indistinguishability

Firstly, we prove that adversary \mathcal{A}

cannot distinguish between the keys generated by deGenKeyPair

and generateKeyPair

.

Theorem 1:

For any probabilistic polynomial-time adversary \mathcal{A}

, |Pr[\mathcal{A}^{deGenKeyPair(\cdot,\cdot)}=1]-Pr[\mathcal{A}^{generateKeyPair(\cdot)}=1]|

is negligible, assuming:

(1) SHA256

is a random oracle.

(2) ECDSA

signatures are unforgeable.

(3) AES

is a pseudorandom function family.

Proof 1:

We use proof by contradiction. We show that if the adversary \mathcal{A}

can distinguish between the keys generated by the two procedures, it must be the case that at least one of the assumptions does not hold.

Suppose that |Pr[\mathcal{A}^{deGenKeyPair(\cdot,\cdot)}=1]-Pr[\mathcal{A}^{generateKeyPair(\cdot)}=1]|

is non-negligible. Since all the randomness comes from prng_{sig}

and prng_{rand}

, it must be the case that \mathcal{A}

can distinguish between prng_{sig}

and prng_{rand}

with non-negligible probability. Recall that prng_{sig}

is a sequence obtained by running AES encryption in the \textit{counter}

mode. So prng_{sig}

is AES_{Enc}(seed, key)

, AES_{Enc}(seed+1, key)

, AES_{Enc}(seed+2, key)

, etc. Let \mathcal{K}

be the set of all possible keys. There are two possibilities: (i) key

is uniformly at random from \mathcal{K}

, which violates our assumption that AES

is a pseudorandom function family. (ii) key

is not uniformly at random from \mathcal{K}

.

Suppose that key

is not uniformly at random from \mathcal{K}

. According to Algorithm 1, key = SHA256(sig)

, where sig=ECDSA_Sign(m, sk)

. There are two cases to consider. (i) \mathcal{A}

knows sig

, which violates our assumption that ECDSA

signatures are unforgeable. (ii) \mathcal{A}

does not know sig

. This violates our assumption that SHA256

is a random oracle.

3.2 Collision-resistance

Theorem 2:

If the probability of collision of keys generated by generateKeyPair

is negligible, the probability of collision of keys generated by deGenKeyPair

is also negligible.

Proof 2:

We use proof by contradiction. Suppose that the probability of collision of keys generated by generateKeyPair

is negligible, and the probability of collision of keys generated by deGenKeyPair

is non-negligible. The adversary can generate a set of keys \mathcal{K}

, and count the number of collisions in \mathcal{K}

. If the number of collisions is negligible, then the keys are generated by generateKeyPair

, otherwise, they are generated by deGenKeyPair

. So the adversary can distinguish between which oracle it is interacting with. But according to Theorem 1, such an adversary does not exist.

3.3 Correctness

Let (e,d,n)

be the keys that \textit{detGenKeyPair}

returns. Let m

be a message, the encryption function, and the decryption function are defined as follows. RSA_{enc}(m) \stackrel{def}{=} m^e

mod n

, RSA_{dec}(m) \stackrel{def}{=} m^d

mod n

.

Theorem 3:

Let (e, d, n)

be any RSA key, which \textit{detGenKeyPair}

returns. For any message m

, RSA_{dec}(RSA_{enc}(m)) \equiv m

mod n

.

Proof 3:

Since l(n) = (p-1)\times(q-1)

and e \times d = 1

mod l(n)

, there must exist some integer k

s.t. e \times d = k \times (p-1) \times (q-1) + 1

. So RSA_{dec}(RSA_{enc}(m)) \equiv m^{e\times d}

mod n \equiv m^{ k \times (p-1) \times (q-1) + 1}

mod n

.

RSA_{dec}(RSA_{enc}(m)) \equiv (m^{ k \times (q-1)})^{p-1} \times m

mod p \equiv m

mod p

, by Fermat's Little Theorem.

Similary, RSA_{dec}(RSA_{enc}(m)) \equiv (m^{ k \times (p-1)})^{q-1} \times m

mod q \equiv m

mod q

, by Fermat's Little Theorem.

According to the Chinese Remainder Theorem, RSA_{dec}(RSA_{enc}(m)) \equiv m

mod n

.

4. Conclusion

Our approach begins by enabling users to sign a fixed message using their ECDSA secret keys, ensuring the unforgeability of these signatures against potential adversaries. Subsequently, we employ SHA256 to hash the generated signatures, producing pseudorandom output, with the assumption that SHA256 behaves as a dependable random oracle. This resulting

hash serves as the key input for the AES block cipher, facilitating the creation of our PRNG. We provide rigorous proof of the security of this construction from indistinguishability, collusion-resistance and correctness.