Introduction To Lamport Signatures

Preface

Read Wikipedia - Lamport Signatures

I am not a cryptographer so if I get anything wrong, please let me know.

There will be **three people** that view this blog post:

- The one that is **new to cryptography** and **will want to read the entire post start to finish**.
  - For you, start from the beginning.
- The one that **understands cryptography** and wants to learn about **Lamport Signatures** and/or **Winternitz-OTS** (unfinished)
  - For you, read the **introduction** and then skip to "How Lamport Signatures Work"
- The one that **understands and solely wants to read about how my implementation works**.
  - For you, skip to "My Reference Implementation of Lamport Signatures"

Introduction

Lamport Signatures, introduced by Leslie Lamport in 1979, are a **One-Time, Post-Quantum, Digital Signature Scheme** that use **Hash Functions** for its Digital Signatures.

Many digital signature schemes have been invented following the publication of Lamport Signatures that follow a scheme similar to Lamport Signatures by using Hash Functions as a way to create digital signatures. An example can be found in RFC8391 (XMSS), a standardized digital signature scheme that uses **Winternitz-OTS+**, an upgraded version of **W-OTS**, which in itself, is an upgraded version of **Lamport Signatures**.

A recent example can be found in the **NIST PQ Round 2 Candidate** SPHINCS+, which is a **stateless hash-based signature scheme** that uses the hashing functions Haraka, SHA256, and SHAKE256.

What Is A Digital Signature?

A **Digital Signature** is almost like a signature in real life. It proves that you signed something, but in the case of Digital Signatures, they use cryptography and mathematics to prove that you signed something using some given **Keypair**, usually with the private key (or sometimes secret key) creating the signature itself.

These **Keypairs** are referred to as:

- **The Public Key**
- **The Private Key** (or Secret Key)

As the names suggests,

- The **Public Key** is Public for all verifiers to check against the **signature** to verify the signature is valid.
- The **Private Key (or Secret Key)** is kept to the prover and used to **generate the digital signature itself** but never revealed
    - an exception to this rule is found in Lamport Signatures as they are labeled One-Time Signature Schemes

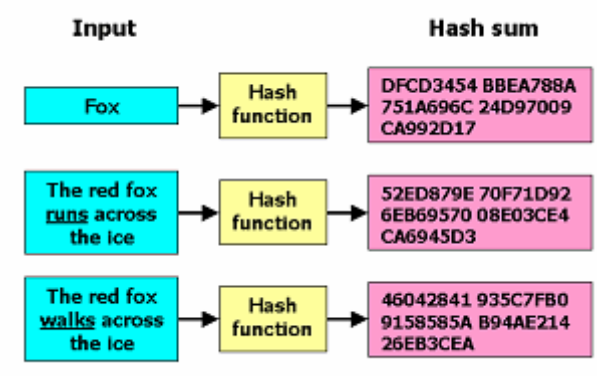
What Is A One-Time Signature

A **One-Time Signature** is a type of signature scheme where using the private key, you can only sign data once. Some digital signature schemes allow multiple signatures from a single private key, allowing you to keep the private key secure and continue signing messages with it.

In a **One-Time Signature**, the **private key is usually revealed after signing** making it only useable for a digital signature scheme once.

What Is A Hash Function?

Read Stackoverflow - Why haven't any SHA-256 collisions been found yet?



The first piece of cryptographic knowledge you need to know is what a **Cryptographic Hash Function** is and what it can be defined as (per wikipedia):

> A **cryptographic hash function** (**CHF**) is a hash function that is suitable for use in cryptography. It is a mathematical algorithm that maps data of arbitrary size (often called the "message") to a bit string of a fixed size (the "hash value", "hash", or "message digest") and is a one-way function, that is, a function which is practically infeasible to invert. Ideally, the only way to find a message that produces a given hash is to attempt a brute-force search of possible inputs to see if they produce a match, or use a rainbow table of matched hashes. Cryptographic hash functions are a basic tool of modern cryptography.

To summarize that, it is a one-way function that takes as input a length of any size and computes it to a fixed length.

Collisions are when two different inputs to the same hash function result in the same output.

In reality, there will always be **infinite collisions** as their are infinite arbitrary sized-inputs, but for practicably, we make assumptions about the hardness of finding collisions, like in **SHA256**, the digest-size is 256 bits, which means there are **2^{256} possible outputs**.

This means that the output will look something like this but it would only be 1s and 0s that represent a large number (2^{256}):

> **SHA256:** 216A6DC4FD320DBCB44E947587442546E7E3C3689BF559BBA7C08D5ADD64685D

As of right now, **no collisions** has never been found for SHA256. Finding a collision would follow the following formula:

I'll just leave this here

> Bitcoin was computing 300 quadrillion SHA-256 hashes per second. That's **300×10^{15} hashes per second**.
>
> Let's say you were trying to perform a collision attack and would "only" need to calculate **2^{128} hashes**. At the rate Bitcoin is going, it would take them
>
> $2^{128}/(300 \times 10^{15} \times 86400 \times 365.25) \approx$ **3.6×10^{13} years**.
>
> In comparison, our universe is only about **13.7×10^{9}**. Brute-force guessing is not a practical option.

How Does Binary and Hexadecimal Work?

Just a quick checker on how binary and hexadecimal works, computers can only read `0` and `1` and so we count in **base 2**, or **binary**. Usually, we count in **bytes**, or 8 bits, a number that can represent **0-255 (2^8)**.

We can also represent them in **hexadecimal**, or **base 16**, using characters `0-9` and `ABCDEF`

Here is `255` in binary, or **2^8**, just ignore the 0b as that just means it represents binary.

> **0b**11111111

Here is `255` in hexadecimal, or **16^2 (2^8)**, just ignore the 0x as that just means it represents hexadecimal.

> **0x**FF

We can **represent larger numbers** like so **(16^4)** or **(2^{16})** which equals `65536`:

> **0x**FF FF

What Is Randomness?

Randomness is a term that can mean a lot of things, but in cryptography, it is mostly suited to mean the randomness that is needed to generate a private key. We can not safely generate a private key without some form of randomness, usually provided by the operating system or by the userspace PRNG. You will see some terms you may not be familiar with. Here are some of them:

- **CSPRNG:** Cryptographically secure pseudorandom number generator
- **PRNG:** Pseudorandom number generator
  - **DRBG**: Deterministic Random Bit Generator
- **RNG:** Random Number Generator

How Lamport Signatures Work

Read [Hash-Based Signatures Part I: One-Time Signatures (OTS)]

Overview

The Basic Idea

The Basic Idea of Lamport Signatures is to have **two secret keys** (x|y) each of 32-bytes (or more) for every bit of an input that we have where:

> 0 = x

> 1 = y

Then, we perform a given **hash function** on both x and y like so to become our public key:

> `h(x)` | `h(y)`

We release this public key and to sign the given message, we only release the secret keys with the corresponding bits in the input.

The verifier can then verify each secret key by hashing it so that it matches the given public key.

Planning Phase

First, you will need to know:

- `n` | The **number of bytes** you would like to **Sign** with your **Keypair**
  - `keypairs` = `n x 8 x 2`
- `d` | the **byte-size of each of your private keys**
  - `d ∈ {32,48,64}`
  - Larger values of `d` result in **larger private key sizes** and **larger signatures** but are more secure

- `hash_function` | the **chosen hash function** you will use to generate the public key and for use in verification
    - Best Practice would be to chose a secure hash function with at least a 256-bit digest size.

Generation Phase

We first need a **CSPRNG** (Cryptographically secure pseudorandom number generator) to generate our private keys for us. These private keys will need to be of the length of 32 bytes (256 bits) or more.

1. Choose the available length of the **input** you would like to sign (`n`) and generate from a **CSPRNG** as many secret keys (`n x 8 x 2`) as required to sign `n`.
    1. Signing 64 bytes (512-bits) will take 1024 keypairs each of `d`-size
2. With the hash function you have chosen, hash each secret key to get the Public Key
    1. `PK` = `h(x)` | `h(y)` for as many secret keys as you have

Signing Phase

To sign a selected input, first convert the input to **binary**. Then for each bit, do the following:

- If the bit is a `0`
    - Publish the secret key for `x`
- If the bit is a `1`
    - Publish the secret key for `y`

And this creates your one-time signature.

Verification Phase

The verifier can then check the signatures by converting the input to binary and like the signing phase:

- if the bit is a `0`
    - Hash the signature to see if it matches the `Public Key (x)`
- if the bit is a `1`
    - Hash the signature to see if it matches the `Public Key (y)`

How Winternitz One-Time Signatures Work

Overview

THIS SECTION IS UNFINISHED

Read RFC8391 - WOTS+

Read Hash-Based Signatures Part I: One-Time Signatures (OTS)

Read Huelsing13

The Basic Idea

Parameters: W-OTS is parameterized by security parameter n, the binary message length m, and the Winternitz parameter $w \in \mathbb{N}, w > 1$ that determines the time-memory trade-off. The last two parameters are used to compute

$$\ell_1 = \left\lceil \frac{m}{\log(w)} \right\rceil, \quad \ell_2 = \left\lfloor \frac{\log(\ell_1(w-1))}{\log(w)} \right\rfloor + 1, \quad \ell = \ell_1 + \ell_2.$$

Moreover, it uses a function chain $\mathcal{C}$. These parameters are publicly known. We can now describe the three algorithms of the scheme.

Public Key = $h^w(x)$

Secret Key = x

Number of Secret Keys depend on `n` and `w`

To prevent a malicious actor from creating fake signatures after releasing our signature, we also have to add and sign a short checksum after the message.

A parameter, known as the **Winternitz Parameter** `w` exists that is used to compute the number of hashing cycles that are completed to create the public key.

`w ∈ {4,16,256}`

`n` | the message length, the length of the private key, public key, or signature element in bytes

The idea is that we can sign **multiple bits** with a single secret key by simply hashing the secret key multiple times, with the Winternitz parameter being a representation of the cycles of hashes we compute. It can be thought of signing in base 2, so:

- 4 cycles signs 2 bits
- 16 cycles signs 4 bits
- 256 cycles signs 8 bits (or one byte)

The size of `w` gives a trade-off between signature size and speed of signing/verification, with a larger value of `w` signing/verifying at a slower-speed but with a smaller signature.

`w = 16` gives the best trade-off between speed and signature size as the value of `w` grows exponentially, as stated in **RFC8391**.

The **hashing function** that is used will also play an important role in both the speed and signature size, with the digest size affecting the signature and public key size, and the hashing function speed affecting the verification/signing speed.

As a basic idea of how signing works, lets say we are using `w = 256` to sign a single byte and that byte is decimal `65` or ASCII `A` represented in binary as:

`0b01000001`

We would first make our public key by signing the secret key 256 times with the hash function, with each output being the next input.

Then, to sign the byte, we would simply hash the secret key 65 times.

The verifier would then hash that until it reaches the public key so the verifier can verify it is right.

My Reference Implementation of Lamport Signatures

View my reference implementation: [Rust] Leslie-Lamport | Docs

Why Do I Believe It Is Secure?

- It is programmed using all safe **Rust**.
- The Private Key Randomness is from the **getrandom** crate which is cross-compatible with the majority of operating systems and gets its cryptographic randomness directly from the **OS CSPRNG**.
- For Public Keys, it can use three hashing algorithms, two of which (**OS_SHA256** and **OS_SHA512**) come directly from the **Operating System's Crypto API**.

TLDR: It's secure because it takes full advantage of the **Operating System's Crypto API**

How To Use

Secure Generation

```
use leslie_lamport::{LamportKeyPair,LamportSignature,Algorithms};

fn main(){
    // Generate Keypair using Operating System SHA256
    let keypair = LamportKeyPair::generate(Algorithms::OS_SHA256);

    // Generate Keypair using Operating System SHA512
    let keypair_sha512 = LamportKeyPair::generate(Algorithms::OS_SHA512);

    // Generate Keypair using Rust Library For Blake2b
    let keypair_blake2b = LamportKeyPair::generate(Algorithms::BLAKE2B);
}
```

More Information

This Rust Library implements the following:

Actions	Description
Generation	Default Generation of 1024 keypairs (x|y) of a default secret key length of 32 bytes (256 bits)
Signing	Can Sign Up To 512-bits of hexadecimal
Verification	Returns Boolean
Serialization	Serializes using **serde**
Deserialization	Deserializes using **serde**
Algorithms Enum	Provides Operating System SHA256 and SHA512 and a Rust Library for Blake2B

For more information, refer below:

- **For the Generation of Private Keys**, the library uses the **Kernel CSPRNG** and creates private keys of size `d` where `d ∈ {32,48,64}` with a default of `d = 32` or `256 bits`
    - This is done through the use of the **Portable, Cross-Platform** getrandom crate as opposed to rand
- **For the Generation of the Public Key**, the choice of three hashing algorithms are given, two of which are wrappers around the Operating Systems Crypto API.
    - **OS_SHA256** and **OS_SHA512** are wrappers around the **Operating Systems Crypto API** using the crypto-hash crate
    - **BLAKE2B** is done through the rust library blake2-rfc with a default digest size of 32 bytes
    - All Hash Functions provide at least a **32 byte (256bit) digest**, with SHA512 providing **64 bytes (512bits)**
- **Can Sign Up To 512-bits (64 bytes)** by default
    - A default of 1024 Private Keys are generated by default of **d-size bytes** where:
        - `d ∈ {32,48,64} bytes` with a default of `d = 32` or `256 bits`
- Remains **Post-Quantum Secure**, including the hash functions chosen.
- **Serialization, Deserialization** implemented through the derivation of Rust's serde.