Fall 2023 CO 485 Final Project

Erica Liu

One-time Signature Scheme and A Python Implementation

1 Introduction

During class we have introduced well-known public-key signature schemes like RSA or DSA, which are based on a trapdoor function: a function acts like a one-way function and its one-wayness is only for parties who do not have access to the private key [1]. Other than this group of public key-based digital signature scheme, one-time signature is simply based on a one-way hash function. As a result, one-time signatures are more efficient with no complex arithmetic involved in key generation and verification. One-time signature was initially developed by Lamport [2] and subsequently enhanced by Merkle[3] and Winternitz.

2 One Time Signature Scheme

Simply speaking, the message signer generates a random number r which serves as a one-time private key. Then signer hashes it though an one-way hash function, h to generate the public key $pk = h(r)$. To sign a message m, the random private key is used to choose from according to the message $\{0, 1\}$ bit, $s = r \oplus m$. When a receiver gets (m, s), if $h(m \oplus h(r)) = s$, the receiver can verify this signature is from the signer.

Algorithm 1 Key Generation

```
Require: Random oracle $R$, one way hash function $H$
Ensure: $pk, sk$
  for j = $0, 1$ do
    for i = $0, \cdots, 255$ do
      $r_i^j \xleftarrow{\$} R$
      $y \leftarrow H(r_i^j)$
      $sk[i][j] \leftarrow r_i^j$
      $pk[i][j] \leftarrow y_i^j$
    end for
  end for
```

2.1 Key generation

A one-time signature scheme involves the generation of a public-private key pair. The public key is used for verification, while the private key is used for signing. Unlike traditional digital signature schemes, the private key in a one-time signature scheme is only valid for a single signature.

For every message, a public key/secret key pair (pk, sk) is generated by having a random matrix sk with size 256×2 and pass it through the random oracle $pk = Hash(sk)$, as stated in Algorithm 1.

2.2 Sign

To sign a message, the user applies a one-time signing algorithm using their private key. This produces a signature that corresponds to the specific message being signed. Once the signing process is complete, the private key becomes obsolete and should never be used again.

Given a message m, the user hashes it using a secure hash function $h_m = Hash(m)$ and gets a binary representation $\{0, 1\}^{256}$. According to 0 or 1 in each position, the user pick element from the private key, as stated in Algorithm 2.

2.3 Verify

The recipient of the message, who knows the public key of the sender, can use the one-time verification algorithm to check the authenticity of the signature. If the signature is valid, it confirms that the message

Algorithm 2 Sign

```
Require: One way hash function $H$, message $m$
Ensure: signature $s$
  for i = $0, \cdots, 255$ do
      if $H(m[i]) == 1$ then
          s[i] = sk[i][1]
      else
          s[i] = sk[i][0]
      end if
  end for
```

was indeed signed by the private key corresponding to the public key, and the message has not been altered.

The recipient receives message and signature pair (m, s) First they hash this message and signature through the same hash function oracle that the sender used. First, they get a binary string, $hm \in \{0, 1\}^{256}$, $hm = Hash(m)$. For each bit, they pick the corresponding value from public key pk, denoted as $pk[hm]$. Then they get a signature hash $sm \in \{0, 1\}^{256}, sm = Hash(s)$. To verify whether the signature is sent from the sender, they can compare whether sm is equal to $pk[hm]$. See Algorithm 3 for pseudocode.

Algorithm 3 Verify

```
Require: One way hash function $H$, message and signature pair $(m, s)$
  for i = $0, \cdots, 255$ do
      if $H(m[i]) == 1$ then
          $pk_i = pk[i][1]$
      else
          $pk_i = pk[i][0]$
      end if
      $hs_i \leftarrow H(s[i])$
      if $pk_i != hs_i$ then return False
      end if
  end for
      return True
```

3 Implementation in Python

```python
import hashlib
import os

def hash_message(message):
    """Hashes a message using SHA-256."""
    return hashlib.sha256(message.encode()).hexdigest()

def key_generation():
    """Generates a key pair for Lamport one-time signature."""
    private_key = [[os.urandom(32) for _ in range(2)] for _ in range(256)]  # Each element is a pair of 32-byte strings
    public_key = [[hash_message(private_key[i][j]) for j in range(2)] for i in range(256)]
    return private_key, public_key

def sign(private_key, message):
    """Signs a message using Lamport one-time signature."""
    if len(message) != 256:
        raise ValueError("Message length must be 256 bits")

    signature = [private_key[i][int(message[i], 16)] for i in range(256)]
    return signature

def verify(public_key, message, signature):
    """Verifies a signature using Lamport one-time signature."""
    if len(message) != 256:
        raise ValueError("Message length must be 256 bits")

    reconstructed_hash = [hash_message(public_key[i][int(message[i], 16)]) for i in range(256)]
    return signature == reconstructed_hash

def run_lamport_signature():
    # Example usage of the Lamport one-time signature scheme
    message = "0123456789ABCDEF"  # 256-bit message in hexadecimal

    # Key generation
    private_key, public_key = key_generation()

    # Signing
    signature = sign(private_key, message)

    # Verification
    is_verified = verify(public_key, message, signature)

    # Output results
    print("Message:", message)
    print("Private Key:", private_key)
    print("Public Key:", public_key)
    print("Signature:", signature)
    print("Verification Result:", is_verified)

if __name__ == "__main__":
    run_lamport_signature()
```

Listing 1: Lamport one-time signature

4 Challenge and Improvement

The Lamport one-time signature scheme implemented above would be weaken the security of the scheme by half after publishing two Lamport signatures using the same key. Merkle introduced an extended scheme to allow signing of arbitrary message, where signatures are embedded in a tree structure to preserve the computation efficiency of one-time signature. [1].

An improvement in security can be achieved by introducing more random private keys, which means we can extend pk, sk from $\{0,1\}^{256\times 2}$ to $\{0,1,\cdots k{-}1\}^{256\times k}$, and extend our hash function H to $H' : \{0,1\}^{256} \rightarrow \{0,1,\cdots,k-1\}^{256}$. Notice that in this case, we need to convert the string from binary representation to k-bit representation, so usually we can take $k = 2^n$ for some n.

4.1 Improved Pseudocode

Similarly we can have the following improved Lamport one-time signature scheme with improved key generation in Algorithm 4, signing in Algorithm 5, and verification 6.

Algorithm 4 Key Generation 2

```
Require: Random oracle $R$, one way hash function $H$
Ensure: $pk, sk$
  for j = $0, 1, \cdots k - 1$ do
      for i = $0, \cdots, 255$ do
          $r_i^j \overset{\$}{\leftarrow} R$
          $y \leftarrow H(r_i^j)$
          $sk[i][j] \leftarrow r_i^j$
          $pk[i][j] \leftarrow y_i^j$
      end for
  end for
```

Algorithm 5 Sign 2

```
Require: One way hash function $H$, message $m$
Ensure: signature $s$
  for i = $0, \cdots, 255$ do
      b ← H(m[i])
      s[i] = sk[i][b]
  end for
```

Algorithm 6 Verify 2

```
Require: One way hash function $H$, message and signature pair $(m, s)$
  for i = $0, \cdots, 255$ do
      $b \leftarrow H(m[i])$
      $pk_i = pk[i][b]$
      $hs_i \leftarrow H(s[i])$
      if $pk_i! = hs_i$ then return False
      end if
  end for
        return True
```

4.2 Improved Python Implementation

For simplicity, we reduce the length of k-bit string to 16 from 256.

```python
import hashlib
import os

```

```python
def hash_message(message):
    """Here we still use SHA-256 for simplicity. The message length is shorten to 16."""
    """Hashes a message using SHA-256."""
    return hashlib.sha256(message.encode()).hexdigest()

def improved_k_key_generation(k,l):
    """Generates a key pair in k bit for Lamport one-time signature."""
    private_key = [[os.urandom(32) for _ in range(k)] for _ in range(l)]  # Each element is a pair of l-length k-bit strings
    public_key = [[hash_message(private_key[i][j]) for j in range(k)] for i in range(l)]
    return private_key, public_key

def improved_k_sign(private_key, message, k, l):
    """Signs a message in k bit using Lamport one-time signature."""
    if len(message) != l*k:
        raise ValueError("Message length must be l * k bits")

    signature = [private_key[i][int(message[i], 16)] for i in range(l)]
    return signature

def improved_k_verify(public_key, message, signature, l):
    """Verifies a signature using Lamport one-time signature."""
    if len(message) != l*k:
        raise ValueError("Message length must be l * k bits")

    reconstructed_hash = [hash_message(public_key[i][int(message[i], 16)]) for i in range(l)]
    return signature == reconstructed_hash



def run_k_lamport_signature():
    # Example usage of the improved k-Lamport one-time signature scheme
    k = 16
    l = 16
    message = "0123456789ABCDEF"  # 16-k-bit message in hexadecimal
    # Key generation
    private_key, public_key = key_generation(k,l)

    # Signing
    signature = sign(private_key, message, k, l)

    # Verification
    is_verified = verify(public_key, message, signature, l)

    # Output results
    print("Message:", message)
    print("Private Key:", private_key)
    print("Public Key:", public_key)
    print("Signature:", signature)
    print("Verification Result:", is_verified)

if __name__ == "__main__":
    run_k_lamport_signature()
```

Listing 2: Improved k-Lamport one-time signature

References

[1] Kemal Bicakci, Gene Tsudik, and Brian Tung. "How to construct optimal one-time signatures". In: *Computer Networks* 43.3 (2003), pp. 339–349. ISSN: 1389-1286. DOI: https://doi.org/10.1016/S1389-1286(03)00285-8. URL: https://www.sciencedirect.com/science/article/pii/S1389128603002858.

[2] L. Lamport. "Constructing digital signatures from a one-way function". In: , *Technical Report CSL-98, SRI International* (1979).

[3] R.C.Merkle. "Secrecy, authentication, and public key systems". In: *Technical report* (1979).