SPHINCS Interim Report

Vedanth Narayanan

March 12, 2016

Introduction

Over the past 7 weeks, we have been focusing on SPHINCS for our project. Most of the work leading up to the past couple weeks have been reading documents that explain the tools being used in SPHINCS. The single biggest challenge for us was primarily getting acquainted with the material. To properly, and throughly, understand SPHINCS we needed to get caught up with a lot of reading. There were multiple papers that required time and dedication to fully understand. Understanding the tools and technologies is crucial if we want to be successful. On top of this, we had the added challenge of figuring out how to piece together the technologies, and how SPHINCS uses them. While we had a lot of catching up to do, we also did get a chance to work on something new. The actual focus is transcribed in this paper. With help from Professor Yavuz, our goal was to come up with a simpler signature scheme that withheld the security that can be incorporated into SPHINCS. The scheme we propose is called Lamport+.

Preliminaries

This section is utilized to briefly talk about existing signature schemes. The sole reason for this is so future references to the specific schemes are not ambiguous.

Lamport Signature Scheme

The Lamport Signature Scheme was created by Leslie Lamport in 1989, and it is the simplest signature scheme that exists. It is also the first One-Time signature that was invented. This scheme makes use of a cryptographic hash function that has already been predetermined.

To put it simply, the idea behind the scheme can be split into two separate pieces. Signer first generates the secret keys, public keys, hashes messages, and gets a signature that is passed off to the verifier. Now, the verifier's job more or less is to follow similar steps so they end up with a similar result. Thus, it can be argued that the message and signature could only come from the signer and no one else. The detailed version of the scheme is mentioned in the Lamport+ section.

WOTS

The primary idea behind the scheme is to break the messages into little blocks, that get processed together, and having an input run through a hash function several times. The

1

number of iterations entirely depends on the message that needs to be signed.
WOTS was built on top of the Lamport signature scheme, and the expectation is for it to be intuitive in its logic, but it's not the case. The complexity of the scheme is heavily influences by figuring out the number of iterations necessary for a value to go through the hash function.

WOTS+

WOTS+ is very similar to WOTS, expect for the addition of XORing random elements every time a value is iterated over hash function. In the key generation phase, WOTS+ generates a set of random numbers that will serve for XORing. Just like the keys are split into chunks, so are the random elements. They get incorporated in the following recursive chaining function

$$c_k^i(x, \mathbf{r}) = f_k(c_k^{i-1}(x, \mathbf{r}) \oplus r_i) \tag{1}$$

Th equation is strictly $i > 0$, but in the case of $i = 0$, $c_0^k(x, \mathbf{r}) = x$. The equation is clever in that it makes sure to XOR different values for every iteration.

Lamport+ Signature Scheme

Lamport+ Signature Scheme is the new scheme we are proposing. It not only brings the simplicity of the original Lamport scheme, but also pulls in elements of the WOTS+ scheme. Our hope is that the original scheme's security is withheld, if not enhanced. Please note that the security of the proposed scheme has not been proven, but it can very well be inferred from the previous. Similar to how WOTS+ introduces XORing of randomized elements to WOTS, the same principle is introduced to Lamport. The following is meant to give an idea of how Lamport+ would work as a One Time Signature, before we extrapolate it to hash chains.

Take note of the following variables:

- $i \in \{0, 1\}$, denotes bit pair
- $n \in \mathbb{N}$, the security parameter
- $s \xleftarrow{\$} \{0, 1\}^m$, seed for the PRNG
- PRNG $g : \{0, 1\}^m \rightarrow \{0, 1\}^n, m < n$
- Cryographic hash function $f_k : \{0, 1\}^n \rightarrow \{0, 1\}^n, k \in \mathcal{K}$

Chaining Function: The chaining function c is $c : \{0, 1\}^{n+m} \rightarrow \{0, 1\}^n$, and it's meant to aid with the Lamport+ signature scheme. The subscript, assuming $x \in \{0, 1\}^n$, indicates a subset of bits from x. $x_{a,b}$ indicates working with bits $x_a, ..., x_b$.

$$\text{Iteration: } c^i(x) = f_k(x_{1,n} \oplus g(x_{n+1,m}))$$
$$\text{Base case: } c^0(x) = x_{1,n} \tag{2}$$

Key Generation

Input: Security parameter n

$$
\begin{aligned}
sk &= ((x_1^0, x_1^1), ..., (x_n^0, x_n^1)) \\
&= ((c_1^0(x)||s_0, c_1^0(x)||s_1), ...(c_n^0(x)||s_0, c_n^0(x)||s_1)) \\
pk &= ((y_1^0, y_1^1), ..., (y_n^0, y_n^1)) \\
&= ((c_1^1(x_1^0)||s_0, c_1^1(x_1^1)||s_1), ...(c_n^1(x_n^1)||s_0, c_n^1(x_n^0)||s_1))
\end{aligned}
\tag{3}
$$

Output: Secret key sk, Public key pk

$$
\begin{aligned}
Gen(1^n) = \; &sk : ((x_0, x_1, s_0||s_1)_1, ..., (x_0, x_1, s_0||s_1)_n), \;\; s_i \xleftarrow{\$} \{0,1\}^m \\
&pk : ((y_0, y_1)_1, ..., (y_0, y_1)_n), \;\; y_i = f_k(x_i \oplus g(s_i))
\end{aligned}
\tag{4}
$$

Signature Generation

$$
Sig(sk, M) = \sigma : (x_{M_1}, s_{M_1})_1, ..., (x_{M_n}, s_{M_n})_n, \;\; M_i \in \{0,1\}
\tag{5}
$$

Signature Verification

Input: Public key of signer pk, hashed message M, and signature σ.

$$
Ver(pk, M, \sigma) = accept \; if \forall i : f(x_i \oplus g(s_i)) = y_i, 1 \le i \le n
\tag{6}
$$

Output: Signature verified if public key is properly derived from signature, if not verification fails.

Key Pair Generation: n is the number of random pairs to be generated. Instead of generating random elements and storing them, a set of seeds will be stored, so a Pseudorandom Number Generator (PRNG) can be used to get values whenever needed. Let PRNG be $g : \{0,1\}^m \rightarrow \{0,1\}^n, m < n$. When passed in a seed of size m, g will return with a pseudorandom number of size n. $s \xleftarrow{\$} \{0,1\}^m$ denotes the seed that will be stored. A unique seed is used for not the key pairs, but the keys themselves. There should be $2n$ seeds total. The secret key is

$$
sk = ((x_0, x_1, s_0||s_1)_0, (x_0, x_1, s_0||s_1)_1, ..., (x_0, x_1, s_0||s_1)_n)
\tag{7}
$$

Each key is n-bits long. Note that there isn't one single seed value s, but one for every key. For every pair, the seeds are concatenated; this idea was borrowed from the paper on XMSS by Buchmann et al. The first half is used for sk_0, while the second half of the seed is used for sk_1. The reasoning behind this becomes apparent when we want to generate our public key. Let $f_k : \{0,1\}^n \rightarrow \{0,1\}^n, k \in \mathcal{K}_n$. The cryptographic hash

function f_k takes in an input of size n, and outputs an n-bit value. Public keys are derived by $pk = f_k(sk \oplus s)$. The whole set of sk is run through the hash function, so we get a bijective public key set. The public key is

$$
\begin{aligned}
pk &= ((pk_0, pk_1)_0, (pk_0, pk_1)_1, ..., (pk_0, pk_1)_n) \\
&= (f_k(sk_0 \oplus g(s_0)), f_k(sk_1 \oplus g(s_1)))_0, (f_k(sk_0 \oplus g(s_0)), f_k(sk_1 \oplus g(s_1)))_1, ..., \quad (8) \\
&\quad (f_k(sk_0 \oplus g(s_0)), f_k(sk_1 \oplus g(s_1)))_n)
\end{aligned}
$$

The seed values that are XORed with the secret keys are indeed seeds, which is why they are run through the PRNG g, so a pseudorandom number is XORed with the secret key. Once the values are XORed, then it can be run through the cryptographic hash function to get our public key.

Signature Generation: The hashed message to be signed is M. Based on every single bit (0 or 1), the corresponding key from a bit pair is selected from the secret keys. These keys make up the signature of the message. For a signature to be properly verified the associated random element needs to be XORed. For the aforementioned reason, the seed for the associated secret key also becomes part of the signature. One may question why the seed is not just stored with the public key, and despite its validity, there is a justification. This is further explained in the Lamport+ Hash Chain section. The Lamport+ signature, as of now, is

$$
\begin{aligned}
\sigma_j &= (sk_i, s_i)_j | i \in \{0, 1\}, j = 0, ..., n \\
\sigma &= (sk_i, s_i)_0, ..., (sk_i, s_i)_n | i \in \{0, 1\}
\end{aligned}
\quad (9)
$$

Signature Verification: The verification for Lamport+ is not as trivial as for Lamport itself. The verifier first obtains the hash of the message (once again, M). Based on the bits of the hash, corresponding keys from public key of the signer are chosen. Extra precaution needs to be taken when the signature is hashed to equal the chosen public key. The Lamport+ signature, along with the secret keys themselves, also have a seed with them. Similar to how the signer derived the public key, the verifier will follow suit. The seed will be run through the PRNG g, and the result will be XORed with the associated sk, and then hashed over f_k. Ideally the values should equal the values that the verifier picked out of the public key. If and when the values don't match is when we know something is wrong, and the signature is not verified.

Lamport+ Hash Chain

Hash chains aid One-Time signatures to be used multiple times with a single key. This can be applied to the Lamport+ signature scheme. The underlining idea here is that after the public key is generated in the scheme, another set of keys are generated based on the previous public key getting hashed. As you can guess, expect the last public key of the chain, all keys before act as a private key.

There are two important things to make note of here. First off, we introduce a new parameter, which we call $l \in \mathbb{N}, l > 0$. This parameter helps keep track of how long the chain is, or how many messages can be signed. Secondly, the public keys will hold new PRNG seeds. These seeds will help produce XOR values for the following set of public keys, and this is not used to verify the signature at that level. This is the reason why the seeds for the signature were not stored with the public key in the previous section.

Note that the idea here is simple, but transcribing it is tricky. The concept is better understood once all the steps are explained. The following is a list of variables to keep note of:

- $n \in \mathbb{N}$, the security parameter
- $l \in \mathbb{N}, l > 0$, the Lamport+ parameter is only known to signer
- Seed $s \in \{0,1\}^m$ is only known to signer until a signature needs to be verified
- PRNG $g : \{0,1\}^m \to \{0,1\}^n, m < n$ is publicly known
- Cryographic hash function $f_k : \{0,1\}^n \to \{0,1\}^n, k \in \mathcal{K}_n$ is also publicly known

Key Pair Generation: n is once again the number of random pairs that will be generated at a time. Again, PRNG will be $g : \{0,1\}^m \to \{0,1\}^n, m < n$. The seed is $s \in \{0,1\}^m$. As a remainder, there will exist a seed for every single key in the chain, so $2nl$ seeds total. The first set of secret and public keys are going to be similar to the previous section. The more interesting thing to look at is the chain itself.

$$
\begin{aligned}
sk_0 &= ((x_0, x_1, s_0||s_1)_0, (x_0, x_1, s_0||s_1)_1, ..., (x_0, x_1, s_0||s_1)_n) \\
&\quad \Downarrow \qquad\qquad\qquad \Downarrow \qquad\qquad\qquad \Downarrow \\
sk_1 &= ((x_0, x_1, s_0||s_1)_0, (x_0, x_1, s_0||s_1)_1, ..., (x_0, x_1, s_0||s_1)_n) \\
&\quad \Downarrow \qquad\qquad\qquad \Downarrow \qquad\qquad\qquad \Downarrow \\
sk_2 &= ((x_0, x_1, s_0||s_1)_0, (x_0, x_1, s_0||s_1)_1, ..., (x_0, x_1, s_0||s_1)_n) \\
&\quad \vdots \qquad\qquad\qquad\quad \vdots \qquad\qquad\qquad\quad \vdots \\
pk &= ((y_0, y_1, s_0||s_1)_0, (y_0, y_1, s_0||s_1)_1, ..., (y_0, y_1, s_0||s_1)_n)
\end{aligned}
\tag{10}
$$

As a note, while the idea of secret keys and public keys are intact here, the notations are not. Much of the keys in the hash chain take on both secret and public key roles.

Signature Generation: The overarching idea here is once a secret key is released, it becomes the public key for the associated secret key before it. The public key that's correlated with the new public key is essentially detached from the hash chain, and will not be used again.

Like mentioned, a hash chain can sign l number of messages. The trick is to use the last set of secret keys to obtain a signature. To sign a message in the following example, our

signature is formed from sk_{l-1}. Once a message is signed, l should be decremented, to prevent discrepancies.

$$
\begin{array}{ccc}
\vdots & \vdots & \vdots \\
sk_{l-2} = ((pk_0, pk_1, s_0||s_1)_0, (pk_0, pk_1, s_0||s_1)_1, ..., (pk_0, pk_1, s_0||s_1)_n) \\
\Downarrow \qquad\qquad \Downarrow \qquad\qquad \Downarrow \\
sk_{l-1} = ((pk_0, pk_1, s_0||s_1)_0, (pk_0, pk_1, s_0||s_1)_1, ..., (pk_0, pk_1, s_0||s_1)_n) \\
\Downarrow \qquad\qquad \Downarrow \qquad\qquad \Downarrow \\
pk = ((pk_0, pk_1, s_0||s_1)_0, (pk_0, pk_1, s_0||s_1)_1, ..., (pk_0, pk_1, s_0||s_1)_n)
\end{array} \tag{11}
$$

The following is what the end of the hash chain would look like after the first message has been signed. The last public key of the chain is detached and ignored. The new public key for the chain is the secret key of the detached public key. By passing off the signature to the verifier at this level, the secret key is given up. To sign another message, the public key will be the current secret key. The chain length, l, will eventually hit 0, and this is when you know you're at the end.

$$
\begin{array}{ccc}
\vdots & \vdots & \vdots \\
sk_{l-2} = ((pk_0, pk_1, s_0||s_1)_0, (pk_0, pk_1, s_0||s_1)_1, ..., (pk_0, pk_1, s_0||s_1)_n) \\
\Downarrow \qquad\qquad \Downarrow \qquad\qquad \Downarrow \\
pk = ((pk_0, pk_1, s_0||s_1)_0, (pk_0, pk_1, s_0||s_1)_1, ..., (pk_0, pk_1, s_0||s_1)_n) \\
pk_{old} = ((pk_0, pk_1, s_0||s_1)_0, (pk_0, pk_1, s_0||s_1)_1, ..., (pk_0, pk_1, s_0||s_1)_n)
\end{array} \tag{12}
$$

This section focused on signature generation in terms of a hash chain. To know how to generate the signatures themselves, refer to the previous section where it is explained in detail.

Signature Verification: Compared to both the Key Generation and Signature Generation phases, this particular phase is the most trivial. The single biggest reason is because the verifier does not know about the hash chain. The verifier only has to verify a signature, and the process is very much similar to verifying an original Lamport signature. Using the hashed message M, the verifier first picks out the public key set before moving on. The verifier needs to remember that every secret key that makes up the signature has a seed value attached to it. This seed is for the PRNG g, which will produce a random value that will get XORed with the key to get the correlated public key. The result should equal what was originally picked out, if not then the signature verification fails.

Security Considerations

Lamport+ Hash Tree

Note that this section hasn't been fully developed yet, and it is still in the works. If Lamport+ is going to get incorporated into SPHINCS, then it needs to be in a tree fashion. The idea we have is that the leafs of a Binary hash tree have the public keys of the hash chains. If there are four leaves, then there are 4 hash chain associated with each of the leaves. The parent node is a hash of the children hash nodes concatenated with each other, and XORed with a set of random elements for every level. The root node serves is the global public key.

When a signature is used from one of the leaves, then the tree needs to be computed again. Note that the whole tree does not need to be recomputed, but only the path to the specific leaf that used a signature. This idea hasn't been thought over entirely yet, but we believe that it should carry over.

Conclusion

In this report, we presented the Lamport+ Signature Scheme, which is much simpler than WOTS and WOTS+. WOTS+ is currently being used in SPHINCS. SPHINCS is indeed stateless, but if Lamport+ were to be integrated instead, the system would become stateful. We went through an example or two to make sure that the scheme worked and it was possible, but have not spent more time on it. We want to go over it again, and see how it fairs against edge cases and such. The plan is also to try to implement it, and integrate it with SPHINCS to see the results.

References

1. Merkle, Ralph C. "A certified digital signature." Advances in Cryptology-CRYPTO'89 Proceedings. Springer New York, 1989.

2. Buchmann, Johannes, et al. "On the security of the Winternitz one-time signature scheme." Progress in Cryptology-AFRICACRYPT 2011. Springer Berlin Heidelberg, 2011. 363-378.

3. Hulsing, Andreas. "W-OTS+ Shorter signatures for hash-based signature schemes." Progress in Cryptology-AFRICACRYPT 2013. Springer Berlin Heidelberg, 2013. 173-188.

4. Bernstein, Daniel J., et al. "SPHINCS: practical stateless hash-based signatures." Advances in Cryptology-EUROCRYPT 2015. Springer Berlin Heidelberg, 2015. 368-397.

5. Buchmann, Johannes, et al. "CMSS - An improved Merkle signature scheme." Progress in Cryptology-INDOCRYPT 2006. Springer Berlin Heidelberg, 2006.

349-363.

6. Buchmann, Johannes, Erik Dahmen, and Andreas Hulsing. "XMSS-a practical forward secure signature scheme based on minimal security assumptions." Post-Quantum Cryptography. Springer Berlin Heidelberg, 2011. 117-129.

7. Mihir Bellare and Phillip Rogaway. Collision-resistant hashing: Towards making UOWHFs practical. In Burton Kaliski, editor, Advances in Cryptology - CRYPTO '97, volume 1294 of Lecture Notes in Computer Science, pages 470-484. Springer Berlin / Heidelberg, 1997. 10.1007/BFb0052256.

8. https://github.com/joostrijneveld/SPHINCS-py

9. https://github.com/CertiVox/MIRACL