SPHINCS Interim Report

Daniel Kirkpatrick
Vedanth Narayanan

March 7, 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 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 the new scheme before we extrapolate it to hash chains.

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 $g : \{0,1\}^m \to \{0,1\}^n | m < n$. When passed in a seed of size m, g will return with a pseudorandom number of size n. $s \in \{0,1\}^m$ 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

$$sk = ((sk_0, sk_1, s_0||s_1)_0, (sk_0, sk_1, s_0||s_1)_1, ..., (sk_0, sk_1, s_0||s_1)_n) \tag{2}$$

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 the following

$$\begin{aligned} pk &= ((pk_0, pk_1)_0, (pk_0, pk_1)_1, ..., (pk_0, pk_1)_n) \\ &= (f_k(sk_0 \oplus g(s_0)), f_k(sk_1 \oplus g(s_1)))_0, (f_k(sk_0 \oplus g(s_0)), f_k(sk_1 \oplus g(s_1)))_1, ..., \\ &\quad (f_k(sk_0 \oplus g(s_0)), f_k(sk_1 \oplus g(s_1)))_n) \end{aligned} \tag{3}$$

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 seed is not associated 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

$$\begin{aligned} \sigma_j &= (sk_i, s_i)_j | i \in \{0,1\}, j = 0, ..., n \\ \sigma &= (sk_i, s_i)_0, ..., (sk_i, s_i)_n | i \in \{0,1\} \end{aligned} \tag{4}$$

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 key itself, also has a seed with it. Similar to how the signer derived the public key, the verifier will follow suit. The seed needs to 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.

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 many times new public keys have been derived. Secondly, the public keys, along with the keys, hold new PRNG seeds. These seeds will help produce XOR values for the following set of public keys, not to verify the signature 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 after going through all the steps.

Key Pair Generation: n is once again the number of random pairs that will be generated at a time. Again, PRNG will be $g : \{0,1\}^m \rightarrow \{0,1\}^n | m < n$. 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.

$$\begin{array}{rl} sk_0 = & ((sk_0, sk_1, s_0||s_1)_0, (sk_0, sk_1, s_0||s_1)_1, ..., (sk_0, sk_1, s_0||s_1)_n) \\ & \Downarrow \qquad\qquad\qquad \Downarrow \qquad\qquad\qquad \Downarrow \\ sk_1 = & ((sk_0, sk_1, s_0||s_1)_0, (sk_0, sk_1, s_0||s_1)_1, ..., (sk_0, sk_1, s_0||s_1)_n) \\ & \Downarrow \qquad\qquad\qquad \Downarrow \qquad\qquad\qquad \Downarrow \\ sk_2 = & ((sk_0, sk_1, s_0||s_1)_0, (sk_0, sk_1, s_0||s_1)_1, ..., (sk_0, sk_1, s_0||s_1)_n) \\ & \vdots \qquad\qquad\qquad \vdots \qquad\qquad\qquad \vdots \\ pk = & ((pk_0, pk_1, s_0||s_1)_0, (pk_0, pk_1, s_0||s_1)_1, ..., (pk_0, pk_1, s_0||s_1)_n) \end{array} \tag{5}$$

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 over arching idea works is after using a secret key, it becomes the public key.

$$\begin{array}{rl} & \vdots \qquad\qquad\qquad\qquad \vdots \qquad\qquad\qquad\qquad \vdots \\ sk_{l-2} = & ((pk_0, pk_1, s_0||s_1)_0, (pk_0, pk_1, s_0||s_1)_1, ..., (pk_0, pk_1, s_0||s_1)_n) \\ & \Downarrow \qquad\qquad\qquad\qquad \Downarrow \qquad\qquad\qquad\qquad \Downarrow \\ sk_{l-1} = & ((pk_0, pk_1, s_0||s_1)_0, (pk_0, pk_1, s_0||s_1)_1, ..., (pk_0, pk_1, s_0||s_1)_n) \\ & \Downarrow \qquad\qquad\qquad\qquad \Downarrow \qquad\qquad\qquad\qquad \Downarrow \\ pk = & ((pk_0, pk_1, s_0||s_1)_0, (pk_0, pk_1, s_0||s_1)_1, ..., (pk_0, pk_1, s_0||s_1)_n) \end{array} \tag{6}$$

The following is what the end of the hash chain would look like. The previous end of the chain is detached and ignored. The new public key for the chain is the secret key of the old 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 will eventually hit 0, and this is when you know you're at the end. Note that this is where the l variable is handy.

$$\begin{array}{rl} & \vdots \qquad\qquad\qquad\qquad \vdots \qquad\qquad\qquad\qquad \vdots \\ sk_{l-2} = & ((pk_0, pk_1, s_0||s_1)_0, (pk_0, pk_1, s_0||s_1)_1, ..., (pk_0, pk_1, s_0||s_1)_n) \\ & \Downarrow \qquad\qquad\qquad\qquad \Downarrow \qquad\qquad\qquad\qquad \Downarrow \\ pk = & ((pk_0, pk_1, s_0||s_1)_0, (pk_0, pk_1, s_0||s_1)_1, ..., (pk_0, pk_1, s_0||s_1)_n) \\ pk_{old} = & ((pk_0, pk_1, s_0||s_1)_0, (pk_0, pk_1, s_0||s_1)_1, ..., (pk_0, pk_1, s_0||s_1)_n) \end{array} \tag{7}$$

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 knows the pubic key, the message, and the signature. This phase is actually exactly the same as from the previous section.

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.

Testing of SPHINCS, RSA, and ECDSA

In this section we will be looking at the performance of SPHINCS, RSA, and ECDSA. These three cryptosystems each sign and verify messages in different ways. Below goes into a quick overview on how each system works. Below is a summary of each the cryptosystems we will be looking at. In the next section we will look at how these three cryptosystems match up against each other.

SPHINCS

SPHINCS makes use of multiple trees to obtain a few time signature (FTS) scheme. This is done using a mixture of Winternitz one time signatures + (WOTS+) and HORS with trees (HORST). The idea of SPHINCS uses a hyper-tree which is borrowed from an idea by Goldreich, which turns a stateful scheme into a stateless scheme. WOTS+ eventually feeds into multiple HORST in the end, which allows for extending signatures to be multi-use instead of single use. SPHINCS has three steps: Key generation, signature generation and signature verification.

RSA

RSA is used for signing messages by using public-key cryptography. It is an asymmetric cryptosystem that is based on the difficulty of factoring two large prime numbers. With RSA there are four steps we go through: key generation, key distribution, encryption and decryption.

ECDSA

ECDSA uses elliptic-curve cryptography to obtain the needed parameters. Key generation in ECDSA requires less bits compared to DSA to obtain the same amount of security. The signature size requires the same amount of bit in both ECDSA and DSA to obtain the same amount of security. ECDSA contains three steps: Key generation, signature generation, and signature verification.

Benchmarks for SPHINCS, RSA, and ECDSA

Benchmarking is done on Arch Linux (4.4.1-2-ARCH), processor: Intel Quad-Core i7-4710HQ 2.50GHz, 8 gigabytes RAM). The testing for SPHINCS is done using a python implementation that is not designed for optimization. This implementation was meant to learn and understand how individual pieces of SPHINCS work [1]. There are optimized versions of SPHINCS available, but compiling the from these implementations are very complex. The testing for RSA and ECDSA is done using MIRACL crypto library, which is an optimized library for performance [2].

The benchmarking is ran using different files sizes, ranging from 10 Kb up to 100 Kb with file sizes being increased in 10 Kb increments. Below you will find two different tests for each cryptosystem, signing and verification.

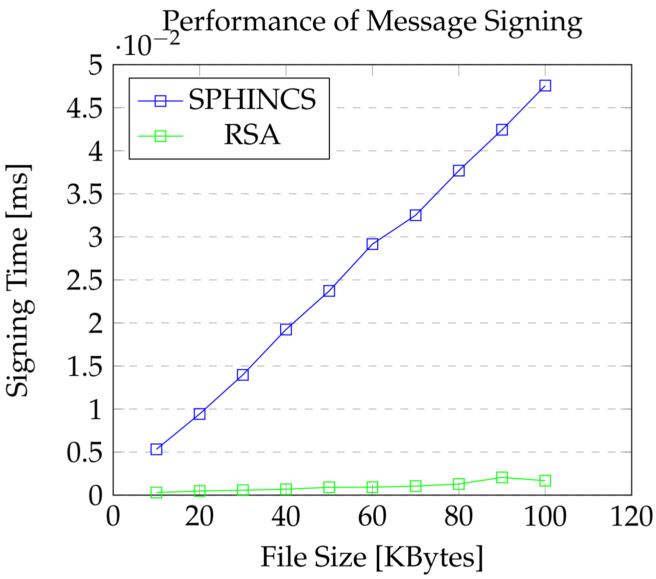
Table 1: Performance of message signing

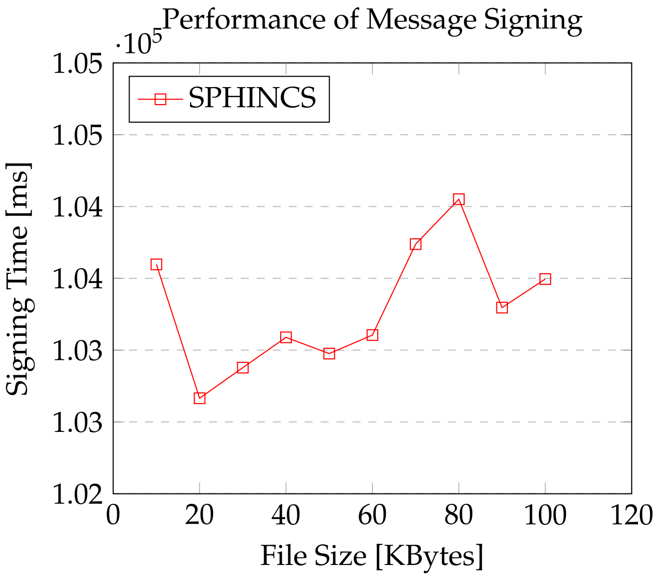
	SPHINCS	RSA	ECDSA
10 Kb	1m 43.597sec	0.005320ms	0.000299ms
20 Kb	1m 42.665sec	0.009426ms	0.000480ms
30 Kb	1m 42.878sec	0.013967ms	0.000574ms
40 Kb	1m 43.089sec	0.019238ms	0.000687ms
50 Kb	1m 42.976sec	0.023722ms	0.000911ms
60 Kb	1m 43.105sec	0.029159ms	0.000921ms
70 Kb	1m 43.738sec	0.032512ms	0.001048ms
80 Kb	1m 44.051sec	0.037693ms	0.001293ms
90 Kb	1m 43.296sec	0.042439ms	0.002060ms
100 Kb	1m 43.495sec	0.047571ms	0.001676ms

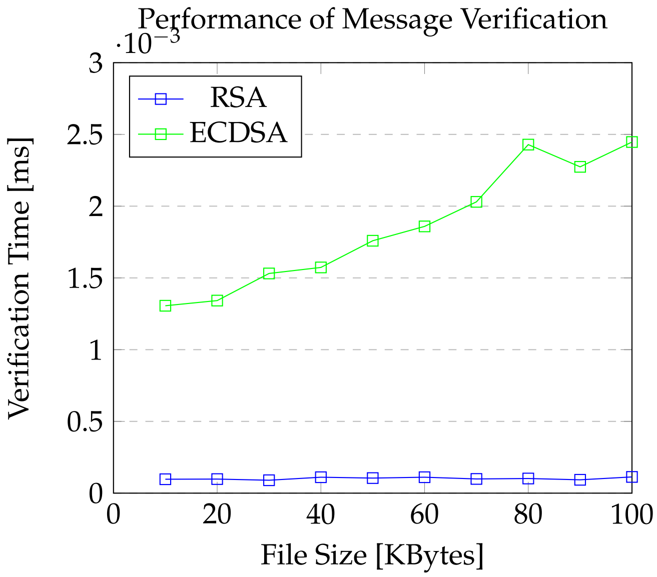
Table 2: Performance of message verification

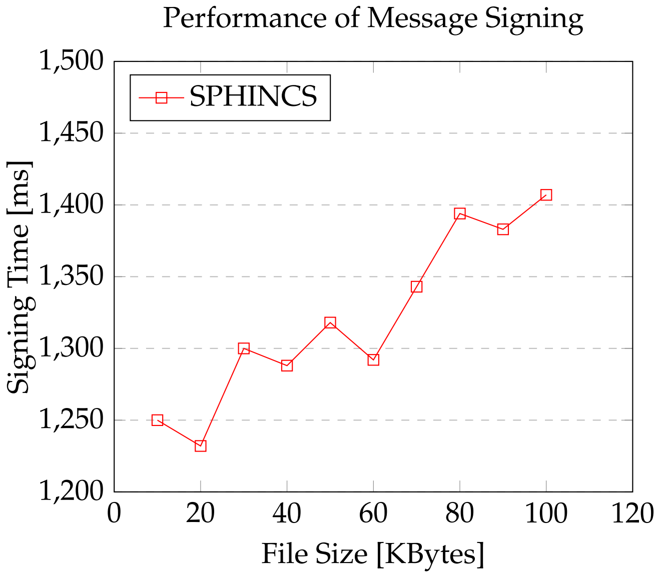
	SPHINCS	RSA	ECDSA
10 Kb	1.250sec	0.000097ms	0.001306ms
20 Kb	1.232sec	0.000098ms	0.001342ms
30 Kb	1.3sec	0.000090ms	0.001531ms
40 Kb	1.288sec	0.000111ms	0.001573ms
50 Kb	1.318sec	0.000105ms	0.001759ms
60 Kb	1.292sec	0.000111ms	0.001859ms
70 Kb	1.343sec	0.000099ms	0.002030ms
80 Kb	1.394sec	0.000102ms	0.002429ms
90 Kb	1.383sec	0.000093ms	0.002274ms
100 Kb	1.407sec	0.000113ms	0.002447ms

Below we will see the data put into another form which might be easier to comprehend the overall performance of each cryptosystem.


Performance of Message Signing
$\cdot 10^{-2}$
5
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
Signing Time [ms]
SPHINCS
RSA
0
20
40
60
80
100
120
File Size [KBytes]


Performance of Message Signing
$\cdot 10^{5}$
1.05
1.05
1.04
1.04
1.03
1.03
1.02
Signing Time [ms]
SPHINCS
0
20
40
60
80
100
120
File Size [KBytes]





NOTE: Due to SPHINCS performance, it was placed into its own graphs. If it were included in the same graphs with RSA and ECDSA, then they would not be visible.

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.
It must stressed, as stated in the Benchmarks section, the SPHINCS implementation that was used for this was not optimized at all. The reason for the use of this implementation of SPHINCS was due to ease of use. There are other optimized implementations available, but the complexity to use them is high.
With that being said, as you can see from the graphs above there are trade-offs with using any of these cryptosystems. As file sizes increase the time it takes to sign and verify messages will also increase.
Based on the implementations used for this benchmark, it would be best to use ECDSA for signatures. ECDSA performs faster than SPHINCS and RSA in terms of signing messages. As you can see though, RSA does verify messages faster than ECDSA, but if you look at the overall performance ECDSA is faster.

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