SPHINCS Interim Report

Daniel Kirkpatrick
Vedanth Narayanan
March 18, 2016

Introduction

We have been focusing on SPHINCS this whole term. Much of the first couple of weeks was reading documentation adn getting up to speed with the subject and field. 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+. On the other hand, we also did some benchmarking to see gauge where SPHINCS stands against RSA and ECDSA.

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 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 $g : \{0, 1\}^m \to \{0, 1\}^n, m < n$
- Cryographic hash function $f_k : \{0, 1\}^n \to \{0, 1\}^n, k \in \mathcal{K}$

Chaining Function

$$\begin{aligned} c^i(x) &= f_k(x_{1,n} \oplus g(x_{n+1,n+m})), i > 0 \\ c^0(x) &= x_{1,n} \end{aligned} \tag{2}$$

The idea for the chaining function, $c : \{0,1\}^{n+m} \rightarrow \{0,1\}^n$, is borrowed from WOTS+. Intuitively, the function just splits the seed from the passed in argument, and uses it to XOR a PRN. The subscript, $x_{a,b}$ defines the specific bits used, i.e. $x_a, ..., x_b$. $a < b \leq (n+m)$ is assumed to be true.

Key Generation
Input: Security parameter n
Output: Secret key sk, Public key pk

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

Signature Generation
Input: Secret key sk, hashed message M
Output: Signature σ

$$\sigma = (\sigma_1, ..., \sigma_n) = (x_1^{M_1}, ..., x_n^{M_n}), M_i \in \{0,1\} \tag{4}$$

Signature Verification
Input: Public key pk, hashed message M, and signature σ
Output: Verification pass or fail

$$pk = (y_1^i, ..., y_n^i) \stackrel{?}{=} (c_1^1(\sigma_1)||\sigma_{n+1,n+m}, ..., c_1^1(\sigma_n)||\sigma_{n+1,n+m}) \tag{5}$$

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 underlying 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 generate XOR values to help produce the following set of public keys. The concept is better understood once all the steps are explained.

Key Generation
Input: Security parameter n

Output: Secret key sk, Public key pk

$$\begin{aligned} sk_k &= ((x_1^{0,k}, x_1^{1,k}), ..., (x_j^{0,k}, x_j^{1,k})) \\ &= ((c_1^k(x_1^{0,k-1})||s_0, c_1^k(x^{1,k-1})||s_1), ..., (c_j^k(x_j^{0,k-1})||s_0, c_j^k(x_j^{1,k-1})||s_1)), \\ &\quad \text{where } s \xleftarrow{\$} \{0,1\}^m, 0 \leq k \leq l, 1 \leq j \leq n \end{aligned} \tag{6}$$

Like you could guess, there will no separate public key that will be generated. If the hash chain is properly generated, this is how it should look:

$$\begin{aligned} sk_0 &= ((x_1^{0,0}, x_1^{1,0}), ..., (x_n^{0,0}, x_n^{1,0})) = ((c_1^0(x)||s_0, c_1^0(x)||s_1), ...(c_n^0(x)||s_0, c_n^0(x)||s_1)) \\ sk_1 &= ((x_1^{0,1}, x_1^{1,1}), ..., (x_n^{0,1}, x_n^{1,1})) = ((c_1^1(x_1^{0,0})||s_0, c_1^1(x_1^{1,0})||s_1), ...(c_n^1(x_n^{0,0})||s_0, c_n^1(x_n^{1,0})||s_1)) \\ &\vdots \\ sk_l &= ((x_1^{0,l}, x_1^{1,l}), ..., (x_n^{0,l}, x_n^{1,l})) = ((c_1^1(x_1^{0,l-1})||s_0, c_1^1(x_1^{1,l-1})||s_1), ...(c_n^1(x_n^{0,l-1})||s_0, c_n^1(x_n^{1,l-1})||s_1)) \end{aligned} \tag{7}$$

Signature Generation
Input: Secret key sk, hashed message M, Lamport parameter l
Output: Signature σ, Updated Lamport parameter l

$$\begin{aligned} \sigma &= (\sigma_1, ..., \sigma_n) = (x_1^{M_1,l-1}, ..., x_n^{M_n,l-1}), M_i \in \{0,1\} \\ l &= l - 1 \end{aligned} \tag{8}$$

Signature Verification
Input: Public key pk, hashed message M, and signature σ
Output: Verification pass or fail

$$pk = (y_1^l, ..., y_n^l) \stackrel{?}{=} (c_1^1(\sigma_1)||\sigma_{n+1,n+m}, ..., c_1^1(\sigma_n)||\sigma_{n+1,n+m}) \tag{9}$$

Lamport+ Hash Tree

The Lamport+ Hash Tree incorporates both the Lamport+ hash chains, and Merkle Hash Trees. The scheme shown here is loosely based on XMSS trees from [6]. The tree allows us sign $\left(\sum_{i=0}^{2^H-1} l_i\right)$ messages, with a single global public key. There are some caveats if the tree is going to be used effectively. Signatures are generated from the leftmost chain to the rightmost. To keep track of what the chain the next signature needs to be generated from, we introduce an index variable i. Once $i = 2^H - 1$, we can determine that signatures for all chains have been used once, and the tree needs to be rehashed using the next set of keys from each of the hash chains. Keep note of the following:

- H : Height of tree

- i : Index of a node starts from the left and goes to right
- $f_k : \{0,1\}^{2n} \rightarrow \{0,1\}^n$
- $N : \{0,1\}^{n+m}$, stands for Node

Key Generation

Begin by choosing an $H \geq 1$ variable first. This will allow us to have 2^H number of Lamport+ hash chains. The hash chains now need to be created to continue following the algorithm. The following equation shows to construct inner nodes. The leaf nodes will be the hashes of the public key of the hash chains initially. The leaves will change once signatures are generated.
Input: 2^H hashed public keys of Lamport+ hash chains
Output: Lamport+ Hash Tree

$$\begin{aligned} N_{i,h} &= f_k((N_{2i,h-1} \oplus g(s_{2i,h-1}))||(N_{2i+1,h-1} \oplus g(s_{2i+1,h-1})))||s_{i,h} \\ &\text{where } 0 \leq i < 2^H, 0 \leq h < H \end{aligned} \tag{10}$$

Signature Generation

A signature is first derived the lamport chain, and the tree is reconstructed to take into account the signature. Every time the tree is reconstructed, a new $s \xleftarrow{\$} \{0,1\}^m$ is generated. A is a list of sibling nodes on the path to the tree root. It's important to remember that once a signature is derived, another shouldn't be generated until the first one has been verified. If there is another generated, then the first signature will not properly verify.
Input: Hashed message M, lamport chain l_i
Output: Authentication path A, signature σ_i, index i

$$\begin{aligned} &\text{if } l_i > 0, \text{ continue, else } i = i + 1, \text{ start over} \\ &\sigma_i = (x_1^{M_1,l-1}, ..., x_n^{M_n,l-1}) \\ &N_{i,h} = \begin{cases} f_k(\sigma_i||(N_{2i+1,h-1} \oplus g(s_{2i+1,h-1})))||s_{i,h} & \text{, left child} \\ f_k((N_{2i,h-1} \oplus g(s_{2i,h-1}))||\sigma_i)||s_{i,h} & \text{, right child} \end{cases} \\ &A = (A_0, ..., A_{H-1}) \end{aligned} \tag{11}$$

Signature Verification

There are two main steps to look out for here. First, verifying the Lamport+ OTS using the signature that is passed in. Secondly, traversing the hash tree to help check the authenticity for the signature. This algorithm was created with help from [6] and [8].
Input: Authentication path A, signature σ, index i, hashed message M, root node of hash tree pk
Output: Verification pass or fail

$$
\begin{aligned}
&sk_l = (y_1^l, ..., y_n^l) \stackrel{?}{=} (c_1^1(\sigma_1)||\sigma_{n+1,n+m}, ..., c_1^1(\sigma_n)||\sigma_{n+1,n+m}) \\
&P_h = \begin{cases} f_k((P_{h-1} \oplus g(s))||(A_{h-1} \oplus g(s))) \oplus s, & \text{if} \lfloor \frac{i}{2^h} \rfloor \equiv 0 \bmod 2; \\ f_k((A_{h-1} \oplus g(s))||(P_{h-1} \oplus g(s))) \oplus s, & \text{if} \lfloor \frac{i}{2^h} \rfloor \equiv 1 \bmod 2; \end{cases}' \\
&\quad \text{where } 0 \leq h < H \\
&P_0 = N_{i,0} \\
&pk \stackrel{?}{=} P_{H-1}
\end{aligned}
\tag{12}
$$

Security Considerations

The idea started by looking at the differences between WOTS and WOTS+. They are very similar, so the difference between them is really what makes WOTS+ more secure than the other. The difference all comes down to the fact that WOTS+ performs an Exclusive OR (XOR) operation before a value is hashed. Our job was to take that idea and apply it to Lamport, and ideally the security would be enhanced.

One of the most *elusive* questions in the project was why the authors of WOTS+ XORed random values before hashing values. Even after thoroughly reading the paper multiple times the reason for XORing random values and the security enhancement are not explained. We found out that the paper on XMSS was written only a couple of years before WOTS+, and the author of WOTS+ also coauthored the XMSS paper. It is in XMSS that the idea for XORing random values came from for WOTS+. This operation, once again, is not properly explained in XMSS. The only reference to the bitmasks points to "Collision-resistant hashing: Towards making UOWHFs practical"[7], written by Mihir Bellare and Phillip Rogaway. It was clear that the XOR trick came from them. They have collectively written multiple papers on collision resistant hashing, and key hashing functions. Upon looking through some of the papers, it seemed to first come about in "Keying Hash Functions for Message Authentication" in 1996. This idea was suggested to them by Adi Shamir, one of the cofounders of RSA. Despite the origins of the idea, the thing to take note of is the variety of ways it gets used.

There are two reasons why random values are XORed in the collision-resistant hashing paper. The first reason being able to bring down the key sizes. The more interesting reason is for key scheduling, especially when it comes to compression functions. From our understanding, setting up keys for compression functions can be tedious, so the XOR operations helps with fixing the keys for the "rounds" in a compression function. The security proof was arduous to understand, but the gist is captured here. Essentially, they prove that by using the XOR trick, the compression function is Target-Collision Resistant (TCR). TCR is second preimage resistant, which essentially the Birthday Paradox.

The following is just a small example. The adversary essentially expects the second collision to be just like the first one, but that is not the case. A new K is introduced in

the first collision, and the only way the adversary will get it is by brute forcing it. The adversary needs something to serve as a base, and the adversary does not get that here.

$$\begin{cases} XLH_{K,K_1,...,K_j}(M_1,...,M_j) & = XLH_{K,K_1,...,K_j}(M'_1,...,M'_j) \\ XLH_{K,K_1,...,K_{j-1}}(M_1,...,M_{j-1}) & \neq XLH_{K,K_1,...,K_{j-1}}(M'_1,...,M'_{j-1})||M'_j \end{cases} \tag{13}$$

Like mentioned, WOTS+ makes use of XORing random values. From our understanding, it didn't need to incorporated at all, however it does make the scheme more secure. While there are small overlapping pieces, like hashing values, we don't see a significant necessity for the trick other than to make the scheme a little more unpredictable.

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.

Problems with SUPERCOP

SUPERCOP[12] is a testing suite that is made by Daniel J. Bernstein and Tanja Lange. It contains multiple different types of crypto systems. The problem I was running into with SUPERCOP is that there is no way to run individual testing without having to completely rip the program apart. Dependencies for the most part are in the folders they should be in, but there are a few which I could not find. I had a hard time trying to figure out how to pull out SPHINCS from SUPERCOP because of this problem. Also since SPHINCS could not be pulled from SUPERCOP, I also couldn't figure out how to pull out ED25519.

Solution to SUPERCOP problem

Eventually I did find an implementation that is based on Erlang. The files it uses are taken from SUPERCOP and the author was able to figure out how to make it work using Erlang[13]. Also not being able to use ED25519, I went with using ECDSA from MIRACL as it's the only ECDSA implementation that I have currently available.

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 Erlang implementation which uses code from SUPERCOP. The authors of the SPHINCS paper created SUPERCOP testing suite which benchmarks all different kinds of crypto systems. This implementation is used as trying to compile individual testing from SUPERCOP is very complicated. The testing for RSA and ECDSA is done using MIRACL crypto library, which is an optimized library for performance [11].

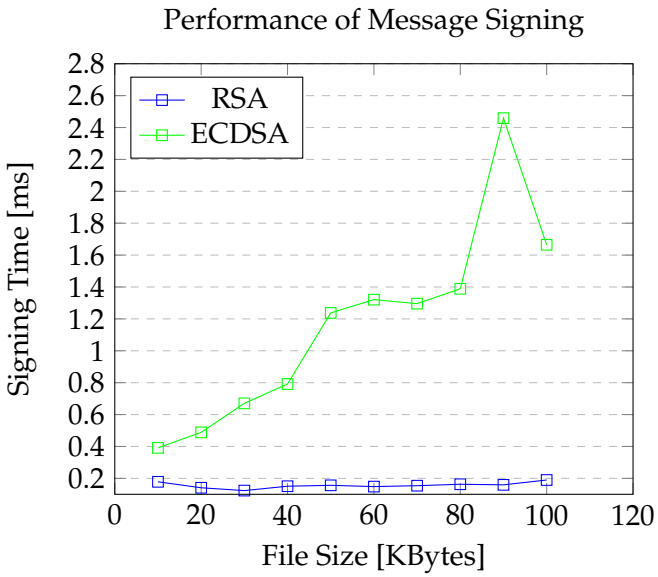
The benchmarking is ran using different files sizes, ranging from 10 Kb up to 100 Kb with file sizes being increased in 10 Kb increments. Below you will find two different tests for each cryptosystem, signing and verification.

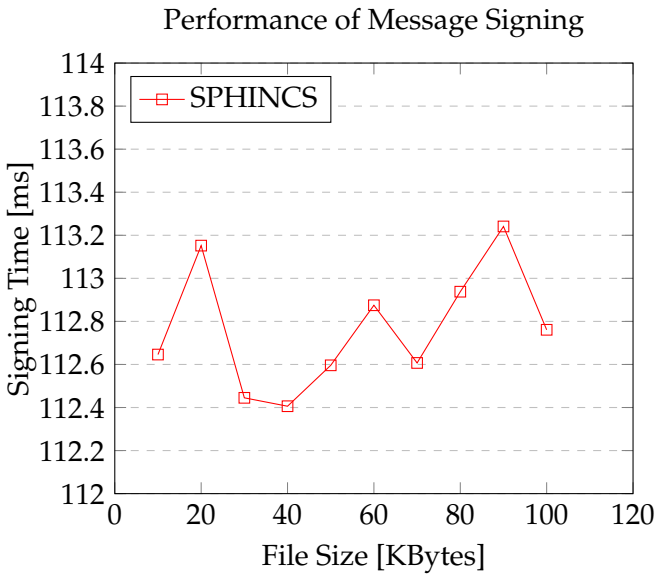
Table 1: Performance of message signing

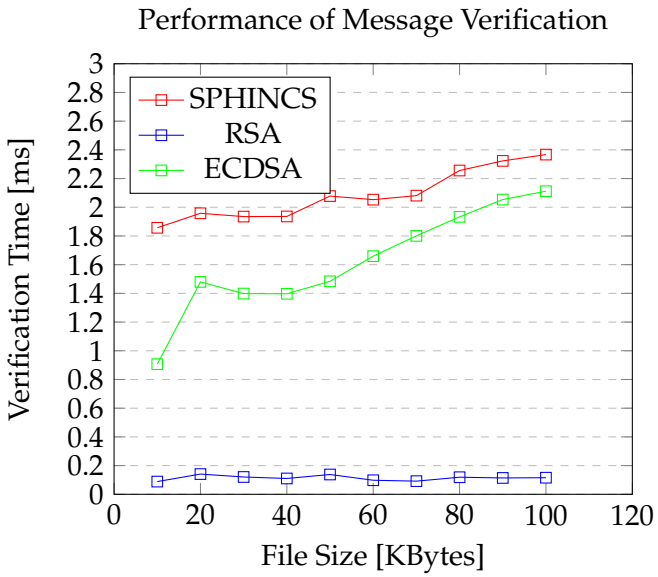
	SPHINCS	RSA	ECDSA
10 Kb	112.646ms	0.178332ms	0.390999ms
20 Kb	113.152ms	0.140999ms	0.488332ms
30 Kb	112.445ms	0.123333ms	0.670332ms
40 Kb	112.406ms	0.150666ms	0.791666ms
50 Kb	112.596ms	0.156332ms	1.237333ms
60 Kb	112.875ms	0.148333ms	1.320665ms
70 Kb	112.607ms	0.153999ms	1.295666ms
80 Kb	112.938ms	0.162999ms	1.388666ms
90 Kb	113.241ms	0.159333ms	2.458666ms
100 Kb	112.761ms	0.189666ms	1.664666ms

Table 2: Performance of message verification

	SPHINCS	RSA	ECDSA
10 Kb	1.857ms	0.088666ms	0.906633ms
20 Kb	1.958ms	0.140999ms	1.479999ms
30 Kb	1.935ms	0.120666ms	1.398666ms
40 Kb	1.936ms	0.110333ms	1.396999ms
50 Kb	2.078ms	0.137999ms	1.483999ms
60 Kb	2.053ms	0.097999ms	1.659333ms
70 Kb	2.081ms	0.091666ms	1.800666ms
80 Kb	2.256ms	0.118999ms	1.932999ms
90 Kb	2.323ms	0.113999ms	2.053666ms
100 Kb	2.367ms	0.115666ms	2.111999ms


Performance of Message Signing
Signing Time [ms]
File Size [KBytes]
RSA
ECDSA
2.8
2.6
2.4
2.2
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
20
40
60
80
100
120


Performance of Message Signing
Signing Time [ms]
File Size [KBytes]
SPHINCS
114
113.8
113.6
113.4
113.2
113
112.8
112.6
112.4
112.2
112
0
20
40
60
80
100
120



NOTE: Due to SPHINCS performance, signature generation was placed into its own graph.

Conclusion

We presented the Lamport+ Signature Scheme along with hash chains and hash trees in this report. The scheme itself is much simpler than both WOTS and WOTS+. The complications arrive when trying to integrate Lamport+ with the Merkle hash trees. Often, hash trees only have one entry for a field, but the nature of a Lamport scheme that's not entirely the case. Based on our algorithm, it is possible to a certain degree, but it's not efficient by any chance. The hash tree algorithm can definitely b more optimized to not only be faster, but costs could be cut doen, and even make a smaller memory footprint. This isn't entirely a promise, but potential future work that would be worth investing time in.

It must stress, as stated in the previous section (Benchmarks), 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, I would suggest using 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. 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. Erik Dahmen, Katsuyuki Okeya, Tsuyoshi Takagi, and Camille Vuillaume. Digital signatures out of second-preimage resistant hash functions. In Johannes Buchmann and Jintai Ding, editors, Post-Quantum Cryptography, volume 5299 of Lecture Notes in Computer Science, pages 109–123. Springer Berlin / Heidelberg, 2008.
9. Johannes Buchmann, L. C. Coronado Garcia, Erik Dahmen, Martin Doring, and Elena Klintsevich. CMSS – an improved Merkle signature scheme. In INDOCRYPT, volume 4329 of Lecture Notes in Computer Science, pages 349–363. Springer, 2006.
10. https://github.com/joostrijneveld/SPHINCS-py
11. https://github.com/CertiVox/MIRACL
12. http://bench.cr.yp.to/supercop.html
13. https://github.com/ahf/sphincs