# Stateless Hash-Based Digital Signature Algorithm

**Documentation**

Simon Hensel[1] and Helena Richter[2]

[1] Albstadt-Sigmaringen University, Albstadt, Germany, hensels1@hs-albsig.de
[2] Albstadt-Sigmaringen University, Albstadt, Germany, richtehe@hs-albsig.de

**Abstract.** The Stateless Hash-Based Digital Signature Algorithms, short SLH-DSA, is a novel, quantum-resistant cryptographic algorithm that utilizes the inherent pseudorandomness of hash functions to create digital message signatures. SLH-DSA improves on previous hash-based systems, such as XMSS, by removing the reliance on state information, which in turn eliminates vulnerabilities related to state management, such as key reuse. This also makes the algorithm more suited to run in distributed environments, by allowing multiple signing operations to be run in parallel, as they are independent of each other. In addition, smaller platforms also benefit from not having to store state information between runs, saving storage space. In this paper, we describe the inner workings of SLH-DSA, as well as document our approach to implementing the algorithm in the languages Python and C.

**Keywords:** SLH-DSA · Post-Quantum Cryptography

## 1 Introduction

Data integrity and authenticity play a vital role in modern cryptography to enable secure communication and digital transactions. Digital Signature Algorithms (DSAs) enable users to validate the origin and integrity of data without the need for direct interaction. The Stateless Hash-based Digital Signature Algorithm (SLH-DSA) is a new DSA that aims to offer a robust and efficient cryptographic solution to ensure data integrity and authenticity, usable by secure, lightweight, and future-proof applications. SLH-DSA is advanced algorithm designed to leverage the inherent strength of hash functions while eliminating challenges associated with stateful signature schemes, such as the eXtended Merkle Signature Scheme (XMSS). As such, SLH-DSA is not vulnerable to errors that can occur with stateful algorithms, such as accidental state reuse. Furthermore, statelessness also reduces the overall complexity of the algorithm, as state management does not have to be taken into account anymore, which also allows for a greater deal of parallelization when running the algorithm.
In this document, we describe the theoretical foundations and inner workings of SLH-DSA, our understanding of the algorithm, how we implement it in the programming languages Python and C and which challenges and solution we encountered doing so. By implementing SLH-DSA, this project contributes to the advancement of quantum-resistant cryptographic applications to provide efficient solution to end users.

## 2 Hash-based Cryptography

Stateless hash algorithms are cryptographic methods that utilize hash functions to ensure security while eliminating the need to maintain state information during operation. In traditional stateful algorithms, maintaining a record of past operations is essential to prevent

vulnerabilities such as key or signature reuse. However, this reliance on state introduces additional complexities to both the algorithm itself as well as respecitve implementations, particularly in distributed or constrained environments.
By contrast, stateless hash algorithms, as the name implies, operate without requiring a persistent state, and instead rely on the pseudorandomness of cryptographic hash functions to sign data. This approach simplifies the algorithm and implementations, as errors associated with state management as well as associated vulnerabilities are of no concern. Stateless approaches are especially valuable in applications like digital signatures, where lightweight, efficient, and secure operations are critical.
Since current, modern hash functions are inherently quite quantum-resistant, assuming a sufficiently large state, signature algorithms that utilize these hash functions can also be considered secure, which ensures their applicability in future-proof cryptographic systems.

## 3 Digital Signature Algorithm (DSA)

The Stateless Hash-based Digital Signature Algorithm (SLH-DSA) is a cryptographic scheme that combines the strength of hash-based security with the simplicity and efficiency of a stateless design. SLH-DSA is a robust alternative to traditional digital signature algorithms, addressing challenges such as state management, accidental key reuse, and scalability in distributed or resource-constrained environments.
At its core, SLH-DSA employs cryptographic hash functions to generate secure, verifiable signatures. Unlike stateful algorithms that require persistent tracking of used keys to remain secure, SLH-DSA eliminates this dependency by deriving keys and signatures dynamically from the pseudorandom images of a hash function to guarantee uniqueness. This stateless approach significantly reduces the operational risks and complexities compared to stateful signature algorithms.
The Algorithm can be divided into 4 Steps:

1. **Key Generation** of private and public key
2. **Key Distribution** of public key
3. **Signature Generation** by sender
4. **Signature Verification** by receiver

During the **Key Generation**, both a private key and a public key are created. The private key consists of two main components: the `SK.seed` which is a random seed used to generate all the secret values for the WOTS+ (Winternitz One-Time Signature Plus) and FORS (Forest of Random Subsets) keys and the `SK.prf`, a pseudorandom function (PRF) key used to generate a randomization value for the message hashing during signature generation. Additionally, the private key includes a copy of the public key components (PK.seed and PK.root) for use during signature generation and verification.
The public key consists of the `PK.seed` which is a public seed used to provide domain separation between different SLH-DSA key pairs. It is also used in many hash function calls. The key also consists of the `PK.root` which is the root of the top-layer XMSS tree, which also serves as the public key of the SLH-DSA hypertree.
The key generation algorithm (`slh_keygen_internal`) takes `SK.seed`, `SK.prf`, and `PK.seed` as inputs. It computes the root of the top-layer XMSS tree (`PK.root`) using the `xmss/_node` function, which recursively constructs the Merkle tree from the WOTS+ public keys. The private key is then formed by bundling `SK.seed`, `SK.prf`, `PK.seed`, and `PK.root`. The public key is formed by bundling `PK.seed` and `PK.root`.
Both `SK.seed` and `SK.prf` must be generated using an approved random bit generator (RBG) with a security strength of at least 8`n` bits, where `n` is the security parameter (16,

24, or 32 bytes depending on the parameter set).

**Key Distribution** in SLH-DSA follows standard cryptographic practices, with a focus on securely sharing public keys while keeping private keys confidential. This ensures that the system remains secure and resistant to attacks, even in a post-quantum setting. Since the public key is not sensitive, it can be distributed freely. Common methods for distributing public keys include Public Key Infrastructure (the public key can be shared via digital certificates issued by a trusted Certificate Authority (CA)), Direct Sharing or Key Servers.

The **signing** process begins with the message (`M`) that needs to be signed, along with the signer's private key (`SK`). To ensure that each signature is unique, even if the same message is signed multiple times, a randomizer (`R`) is generated. This is done using the `PRFmsg` function, which takes `SK.prf`, a previously generated random value `addrand`, and the message `M` as inputs. For the deterministic variant of the algorithm, `addrand` is replaced with `PK.seed`. The randomizer is then combined with the message, `PK.seed`, and `PK.root` using the `Hmsg` function to produce a message digest. This digest serves as the basis for the rest of the signing process.
The message digest is divided into different parts to determine how the signature will be constructed. One portion of the digest is used to select a FORS key from the large set of FORS key pairs in the SLH-DSA key structure. Another portion is used to determine the leaf index within each FORS tree. This ensures that the signature is tied to a specific FORS key.
Once the FORS key is selected, it is used to sign the first part of the message digest. This involves generating a FORS signature using the `fors_sign` function. The FORS signature includes secret values from the FORS private key, along with their corresponding authentication paths in the FORS Merkle trees. These authentication paths allow the verifier to reconstruct the FORS public key during verification.
After the FORS signature is created, the corresponding FORS public key is computed. This public key is then signed using the hypertree structure, which consists of multiple layers of XMSS trees. The `ht_sign` function generates a hypertree signature by creating a sequence of XMSS signatures. The process starts at the bottom layer of the hypertree, where the FORS public key is signed, and moves up through each layer until the top layer is reached. Each layer signs the public key of the layer below it, ultimately producing a chain of signatures that authenticate the FORS public key. The public key of each layer gets signed with a WOTS+ signature, before the next layer is processed.
The final SLH-DSA signature is composed of three main components: the randomizer (`R`), which ensures the uniqueness of the signature, the FORS signature, which signs part of the message digest and the hypertree signature, which authenticates the FORS public key through the layered XMSS structure.

The **verification** process starts with three key inputs: the original message (`M`), the signature (`SIG`), and the signer's public key (`PK`). The signature consists of three components: the randomizer (`R`), the FORS signature, and the hypertree signature. The public key includes `PK.seed` (a public seed) and `PK.root` (the root of the top-layer XMSS tree), which are essential for the verification process.
The verifier starts by recomputing the message digest using the same process that the signer used during the signing process, as described above. This involves hashing the message `M` together with the randomizer `R`, `PK.seed`, and `PK.root` using the `Hmsg` function. The resulting digest should match the one used by the signer to generate the signature. If the digest does not match, the signature is immediately invalid.
The recomputed message digest is split into parts to determine the specific keys and trees

used in the signing process. One portion of the digest is used to identify the FORS key that was used to sign part of the message. Another portion determines the leaf index of each tree in the FORS structure. These indices are important for reconstructing the FORS public key and verifying the hypertree signature.
Using the FORS signature and the relevant portion of the message digest, the verifier computes a candidate FORS public key using the `fors_pkFromSig` function. This involves reconstructing the Merkle tree roots from the FORS signature and the message digest. The verifier then hashes these roots together to produce the candidate FORS public key. This step ensures that the FORS signature corresponds to the correct FORS key and that the message digest was signed properly.
The candidate FORS public key is then verified using the hypertree signature. The hypertree signature consists of a sequence of XMSS signatures, starting from the bottom layer of the hypertree (which signs the FORS public key) and moving up to the top layer (which signs the XMSS public key of the layer below). Again, each signature is singed by a WOTS+ signature chain. The verifier uses the `ht_verify` function to check each XMSS signature in the sequence. For each layer, the verifier reconstructs the corresponding XMSS public key using the `xmss_pkFromSig` function and compares it to the expected value. This process continues until the top layer is reached.
At the top layer of the hypertree, the verifier compares the computed XMSS public key to the `PK.root` from the signer's public key. If the two values match, the signature is considered valid. This confirms that the signature was generated using the signer's private key and that the message has not been altered. If the computed public key does not match `PK.root`, the signature is invalid, indicating either tampering with the message or an incorrect signature. [oST24]

SLH-DSA is especially useful for modern applications where lightweight and scalable solutions are critical. It is resilient to common vulnerabilities found in stateful systems and offers robust security against evolving threats, including those posed by quantum computing. Furthermore, its reliance on well-studied cryptographic hash functions ensures that it remains a practical and secure choice for both current and future cryptographic needs.

### 3.1 Parameter Sets

SLH-DSA comes with different parameter sets that govern different aspects of the algorithm, including bit security and signature length. Parameter sets are divided into three security categories, indicated by the parameter `n`, which determines the length of a hash output in bytes, so 16, 24 and 32 bytes representing 128, 192 and 256 bit security, respectively. In addition, each security category has two different parameter sets each, suffixed by the letters `f` or `s`, which stand for `"fast"` and `"small"`. These achieve a tradeoff between program execution time and storage space: the `"fast"` option comes with a shorter execution time but larger signatures, whereas the `"small"` option outputs significantly smaller signatures at higher computing cost. The `"small"` set achives shorter signatures by using a smaller hypertree (parameter `d`) and fewer FORS trees (parameter `k`), whereas the `"fast"` implementation comes with shorter authentication paths (parameters `h'` and `k`), saving processing time.

## 4 Implementing the Algorithm

First, we implement a prototype of SLH-DSA in Python. This has numerous advantages: First, Python code is rather easy to run and debug since no compilation is needed and Python's stacktraces are really detailed. Second, the whole signature algorithm can be implemented using only the Python standard library. After the Python protoype is confirmed to be working, we implement the algorithm again in the programming language C, which offers vastly faster execution times compared to Python. With this approach, we can always make sure the C implementation is working as intended by comparing it with the Python protoype. For both implementations, we choose to use one dedicated file per chapter in the standard specification, so the code follows the overall structure of the specification paper. Note that only the parameter sets using SHAKE are implemented. Both implementations are tested with the official test vectors provided by the NIST on their official GitHub page [usn24].

## 4.1 Python

The Python implementation is rather straightforward. We start with chapter 4 of the specification, which governs the pseudo-random functions and hash addresses. All functions regarding the hash address `ADRS` are grouped up in a class of the same name, all other functions use regular methods. The parameter sets are provided via a global variable that gets initialized at the beginning of the program. Altough global variables are considered bad practice in general, we deliberately take this approach as now the parameters do not have to be passed to every function call individually. In addition, the parameters never change during runtime and are treated as read-only, so we do not have to take race conditions or similar issues into account.
The Python implementation always operates on `byte`-objects: the initial message, as well as the key values and the hash outputs are bytes. All pseudo-random functions are implemented using Python's standard module `hashlib` to generate the various hashes. Furthermore, the `secrets` module is used to generate cryptographically secure random values during the key generation and for the non-deterministic variant of the algorithm.
The rest of the algorithm pretty much follows the pseudocode provided in the specification. In cases where floor division is decessary, we use Python's floor operator `//`, if the ceiling of a value has to be calculated, we add the $denominator - 1$ to the numerator. For instance, $\lceil \frac{k*a}{8} \rceil$ is implemented as `(k * a + 7) // 8`.

## 4.2 C

The C code now gets implemented after the Python protoype is finished and confirmed to be working. We take the same approach of using one code file per chapter, which in the context of C means one `.c` file accompanied by a `.h` file with the same name. Here, all parameters are stored in a struct, that gets initialized at the beginning of the program and is passed to each function call individually, so we no longer have to deal with a global variable. Similarly, the address `ADRS` is now also stored in a struct, since C does not support classes.
The code operates on byte arrays, similar to Python's byte-objects: the message that gets signed, as well as the keys, the hash outputs and the signatures are stored in byte arrays. As such, every function that returns a byte value, has an additional input called `buffer`, which is a pointer to an array where the output values get copied to via the `memcpy()`-function. Great care has been taken to avoid any form of buffer overflows or overreads when doing so. Furthermore, we intentionallt do not use `malloc()` or `calloc()` to avoid problems such as memory leaks when forgetting to call `free()` or segmentation faults when trying to access memory that has already been free'd.
Ceiling and floor division is implemented in the same way as the Python protoype, since the math library of C returns values of data type `double` when calling `ceil()` or `floor()`, which need to be converted back to an integer type. While this is generally possible, the larger parameter sets, namely `SLH-DSA-SHAKE-192s` and `SLH-DSA-SHAKE-256s`, operate on values that are 64 bit in size, which causes floating point exeptions when trying to cast

from a 64 bit double to a 64 bit integer.
To further optimize performance, we use `x & 1 == 0` instead of `x % 2 == 0` to check if a given number is even, as bit operations are always faster than division or multiplication. Similarly, we replace multiplications and divisions by the power of 2 with bit shifts, so $x * 2^y$ is implemented as `x << y`, whereas $x/2^y$ becomes `x >> y`.
The pseudo-random hash functions are first implemented using the `libgcrypt` library, a cryptographic library readily available on most Linux systems. In a second iteration, we replace SHA-256 from the `libgcrypt` library with the `kcp/optimized1600AVX512` implementation of SHA-256, which is part of the SUPERCOP cryptography benchmark system, to reduce the program's exeption time. `kcp/optimized1600AVX512` is optimized for processors that support the AVX512 instruction set, which includes our development machines, which should provide a performce boost when executing the code on these platforms. Random numbers are generated using the cross-platform `sodium` library.
While not impacting performance, we also take measures to modernize the C codebase. For instance, we use the datatypes defined in `stdint.h`, so `unsigned char` becomes `uint8_t`, `unsigned int` becomes `uint32_t` and so on. In addition, we replace header include guards with the simple macro `#pragma once`, which achives the same effect without having to think about our own definitions.

## 4.3 Performance

As mentioned above, in a later development iteration, we replace the `libgcrypt` library with the `kcp/optimized1600AVX512` implementation of SHAKE256 in hopes of increasing performance. In this section, we run some performance tests using both libraries to compare the two. To measure performance, we use the perf program [1], that is part of the Linux kernel to measure CPU cycles. The SLH-DSA code is always compiled with the same compiler options, mainly `-O3` and `-march=skylake-avx512`.
For our tests, we generate a keypair, sign the message with the private key and validate the resulting signature with the public key using all 6 implemented parameter sets. This also explains the high clock cycle count, as each iteration consists of 6 key generations, 6 signature generations and 6 signature verifications. This process gets repeated 100 times, to obtain a sufficiently accurate performance average. For all tests, the message to sign and the context string stay the same, while the keys are randomly generated. The results can be see in Table 1, with the `kcp/optimized1600AVX512` implementation achieving a performance increase of 32.95%.

**Table 1:** Average CPU clock cycles per test run.

| **SHAKE256 Implementation** | **Avg. CPU Cycles per Test** |
|---|---:|
| `libgcrypt` | $75,011,809,215$ |
| `kcp/optimized1600AVX512` | $50,297,966,938$ |
| **Performance increase** | 32.95% |

A vast majority of the algorithm's runtime is used on hashing, as can be seen in Table 2, which shows an excerpt of select functions and how many CPU cycles they take to complete during one test run. The functions names for SHAKE256, prefixed with `KeccakP1600...` stem from the `kcp/optimized1600AVX512` implementation, while the other functions names follow the pseudocode from the standard and are implemented by us.

These results show that the implementation of the hash function poses a heavy bottleneck on the programs runtime. A lot of execution time, and by extension, power

[1]https://perfwiki.github.io/main/

**Table 2:** Percentage of all CPU cycles by function.

| % of Clock Cycles | Function Name |
|---:|---|
| 85.93% | `KeccakP1600_Permute_24rounds` |
| 6.30% | `KeccakP1600_AddBytes` |
| 2.41% | `KeccakWidth1600_Sponge` |
| 2.03% | `F` |
| 0.34% | `PRF` |
| 0.32% | `fors_node` |
| 0.22% | `KeccakP1600_Initialize` |
| 0.22% | `wots_pkGen` |
| 0.14% | `H` |
| 0.14% | `KeccakP1600_ExtractBytes` |
| 0.02% | `xmss_node` |
| 0.01% | `KeccakF1600_FastLoop_Absorb` |

consumption, can be saved by choosing an implementation of SHAKE256 that is properly optimized for the target platform where the algorithm should be run.

## 5 Conclusion

This last section details our learnings from implementing SLH-DSA. First it is always important to properly read the documentation, as many errors or inconsistencies can be avoided when staying close to the source. Next, it is really helpful to test individual components of the code, before implementing the next module. This avoids subsequent errors and makes debugging easier, as faults in a component that gets called very late during runtime can be hard to debug and find. In that vein, it is also really helpful to compare your own code base with other, already existing implementations. This helped us test and debug individual components, as well as alternative solutions to coding problems. All in all, this project really helped to provide a perspective on how cryptographic algorithms are implemented, how they work and the common pitfalls that can happen during development. The coding practice also helped solidify our knowledge in both Python and C regarding best practices and coding techniques.

## References

[oST24] National Institute of Standards and Technology. Stateless Hash-Based Digital Signature Standard. Technical Report NIST.FIPS.205, 8 2024.

[usn24] usnistgov. ACVP-Server, https://github.com/usnistgov/ACVP-Server, 11 2024.