Stateless Hash-Based Digital Signature Standard

Dokumentation

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. In this paper we prove that the One-Time-Pad has perfect security. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Donec odio elit, dictum in, hendrerit sit amet, egestas sed, leo. Praesent feugiat sapien aliquet odio. Integer vitae justo. Aliquam vestibulum fringilla lorem. Sed neque lectus, consectetuer at, consectetuer sed, eleifend ac, lectus. Nulla facilisi. Pellentesque eget lectus. Proin eu metus. Sed porttitor. In hac habitasse platea dictumst. Suspendisse eu lectus. Ut mi mi, lacinia sit amet, placerat et, mollis vitae, dui. Sed ante tellus, tristique ut, iaculis eu, malesuada ac, dui. Mauris nibh leo, facilisis non, adipiscing quis, ultrices a, dui.

Keywords: Something · Something else

1 Introduction

In the realm of modern cryptography, ensuring data integrity and authenticity is a cornerstone of secure communication and digital transactions. Digital Signature Algorithms (DSAs) are pivotal in this context, enabling entities to validate the provenance and integrity of data without the need for direct interaction. Among the diverse array of DSAs, the Stateless Hash-based Digital Signature Algorithm (Stateless Hash-DSA) emerges as a robust and efficient cryptographic solution tailored for secure, lightweight, and future-proof applications.
This project aims to explore and implement the Stateless Hash-DSA, an advanced algorithm designed to leverage the inherent strength of hash functions while addressing challenges associated with stateful signature schemes. Unlike traditional approaches that rely on maintaining and managing state information, Stateless Hash-DSA eliminates the complexities of state management, thereby reducing operational risks such as accidental state reusea common vulnerability in stateful systems.
By implementing the Stateless Hash-DSA, this project contributes to the advancement of secure digital signature techniques suitable for environments where simplicity, efficiency, and resistance to quantum adversaries are crucial. The following documentation outlines the theoretical foundations, implementation details, and practical applications of the Stateless Hash-DSA.

2 Hash-based Cryptography

Stateless hash algorithms are cryptographic methods that leverage 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 can introduce operational complexities and risks, particularly in distributed or constrained

environments.
By contrast, stateless hash algorithms operate without requiring a persistent state, relying solely on the cryptographic strength of hash functions to secure data. This design simplifies implementation, reduces the risk of errors associated with state management, and enhances resilience against attacks that exploit state inconsistencies. Stateless approaches are especially valuable in applications like digital signatures, where lightweight, efficient, and secure operations are critical.
Stateless hash algorithms are also well-suited for post-quantum cryptography, leveraging the inherent robustness of hash functions against quantum adversaries, ensuring their applicability in future 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 or states to ensure security, SLH-DSA eliminates this dependency by deriving keys and signatures dynamically in a manner that guarantees their uniqueness and integrity. This stateless approach significantly reduces the operational risks and complexities associated with managing and safeguarding state information.

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

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 enhanced 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.
This algorithm is a key step toward creating secure, efficient, and future-proof digital systems, making it a compelling choice for developers and security professionals seeking innovative cryptographic solutions.

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. 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 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 mimicks the overall structure of the specification paper. Both implementations are tested with the official test vectors provided by the NIST on their official GitHub page [1].

4.1 Python

The Python implementation is rather straightforward. We started with chapter 4 of the specification, which governs the pseudo-random functions and hash addresses. All functions regarding the hash address are grouped up in a class, 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, so we do not have to take race conditions or similar issues into account.
The Python implemented 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 $\mathit{denominator} - 1$ to the numerator. For instance, $\lceil \frac{k*a}{8} \rceil$ becomes `(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 whose address passed to each function call individually. Similarly, the address is now also a struct, since C does not support classes.
The code operates on byte arrays, 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. Great care has been taken to avoid any form of buffer overflows or overreads when doing so. Furthermore, we do not use the `malloc()` 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`, produce values that are 64 bit in size, which causes floating point exeptions when trying to cast from a double to a 64 bit integer.
The pseudo-random hash functions are first implemented using the `libgcrypt` library, which is available on many Linux platforms. In a second iteration, we replace `libgcrypt` with the `kcp/optimized1600AVX512` implementation, which is part of the supercop cryptography benchmark system, for the pseudo-random functions. Random numbers are generated using the cross-platform `sodium` library.

[1] https://github.com/usnistgov/ACVP-Server

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 make our own definitions.