IJHSR
ijhighschoolresearch.org
RESEARCH ARTICLE

# Certificate-Based Key Replacement in Single Use Hash-Based Signatures: A Post-Quantum Technique

Matthew D. Grupenhoff

Henry M. Gunn High School, Palo Alto, California, 94306, USA; mg27713@munge.com
Mentor: Dr. Dong, Polygence

ABSTRACT: The recent advancement in quantum computing poses a concern over the security of current public key cryptography systems. Since then, hash-based signature schemes have been created to replace vulnerable schemes, although they have limitations such as requiring a large key and/or signature size or only working a limited number of times. This paper discusses a solution to these two problems through key replacement to allow a single initial agreement to be used an arbitrary number of times with a reasonable key size. The experiments were done by extending the Lamport signature scheme with this key replacement, leading to a signature scheme that is slower than DSA and ECDSA, but faster than RSA for sufficiently large security levels. As a result, while applying this technique does not lead to very fast performance, it still has the advantage of post-quantum security, and it is not excessively slow.

KEYWORDS: Systems Software; Cybersecurity; Digital Signatures; Post-Quantum; Hash-Based.

## Introduction

The current internet is secure because of cryptographic systems such as digital signatures. However, the recent advancement in quantum computing poses a concern over the security of current public key cryptography systems. This is the result of an algorithm from 1994 known as Shor's algorithm, which is capable of solving the discrete logarithm problem as well as factoring large numbers, both in polynomial time.[1,2] As a result, the public key cryptography used every day, such as transmitting private information online, will eventually be rendered completely insecure.

Today's internet uses a number of concepts from cryptography to remain secure. A key concept in cryptography is encryption, which is the transformation of secret information into a form that can only be transformed back if a small secret referred to as a key is known; this reverse process is known as decryption. To distinguish between encrypted data from different parties, a key is also required to encrypt the data, to begin with. This key is important because it ensures that even if an attacker knows how the security works, they still need additional information that cannot be efficiently discovered.[3]

Some algorithms, known as symmetric algorithms, use the same key for both encryption and decryption, which is generally easy to implement; they rely on the fact that certain functions, together with their inverses, can be trivially computed if a piece of secret information is known. In cryptography, this secret information is stored as the key. While symmetric cryptography is generally very secure, there are times when it cannot be used, such as allowing anyone to verify the authenticity of a message while restricting who can create authentic messages.

The security of a cryptographic algorithm is defined by its security level n, which is the number such that computing unintended information, such as a private key from a public key, requires a maximum of $2^n$ comparisons;[4] therefore, it requires $2^{n-1}$ comparisons on average. Security levels are helpful for comparing the performance of new algorithms with existing algorithms, and deciding what parameters to use when switching algorithms.

One limitation of symmetric algorithms is that they cannot be used to prevent parties from decrypting data while allowing them to encrypt it, or vice versa. By using an algorithm that uses a separate key for each direction, this problem is solved; such algorithms are known as asymmetric algorithms. One of the two keys, known as the public key, is typically shared to allow anybody to either encrypt or decrypt data, depending on what the encryption is being used for, while the other key, known as the private key, is not shared with the public, and is typically only used by the party that generated it; the private key is required to use a function known as a trapdoor function, which is used for the operation that is not intended to be used publicly. These keys must be generated through a method such that the private key cannot be efficiently computed from the public key; if it is possible to derive the private key from the public key, then the security is effectively reduced to that of a symmetric algorithm.

One particular algorithm, RSA (Rivest, Shamir, and Adleman), has been used for decades due to it receiving no critical attacks. Its key generation algorithm starts by generating 2 random prime numbers, known as p and q, and multiplying them, releasing the product as part of the public key. This is significant because there is no efficient algorithm for finding factors of a large number on a classical computer but being able to efficiently find factors would mean p and q are recoverable. Therefore, the private key can be computed. This defeats the security of RSA because, with the private key, an attacker can take unauthorized actions such as forging secure messages.

It is also useful at times to be able to verify a piece of data without storing it in its entirety, either because it is inefficient



DOI: 10.36838/v5i4.14

or insecure to store the entire piece of data. For this, a one-way function is also used to transform a variable-length input into a fixed-length output with an extremely low probability of two distinct inputs with the same output being discovered. These one-way functions are known as hash functions, and their outputs, known as hashes, are often stored next to critical data to ensure it was not accidentally modified or used for cases such as password storage. In both of these cases, the data can be verified by taking the hash a second time and comparing the result, but there is an extremely low probability of the hash being reversed.

Sometimes, verifying a large piece of data at once is not an option. In these cases, the data can be verified in smaller parts by hashing each part, then hashing the list of hashes and storing that master hash securely. As this list grows, multiple master hashes can be evenly divided with a higher master hash, and this tree can be extended as deep as necessary. A tree like this is known as a Merkle tree, or a hash tree.

One significant application of asymmetric cryptography is the concept of digital signatures, which can be verified by anybody but only generated by the person with a private key. Given an asymmetric encryption scheme, such as RSA, and a hashing algorithm, signatures are commonly implemented by hashing the content and encrypting the hash with the private key; the signature can then be verified by decrypting it with the public key and comparing the result with the hash of the content.

Quantum computers are capable of operating on quantum states of 0 and 1, as opposed to how everyday computers, known as classical computers, operate solely on 0 and 1. This allows them to try every possible input to a one-way function at the same time. However, due to limitations imposed by quantum mechanics, a special circuit is needed at the end to extract any useful results.

Shor's algorithm is an algorithm for quantum computers capable of finding factors of a number n in $O(logn)^2\ log(logn)\ log(log(logn)))$ time, which is far more efficient than the $O\ (x^{(logn)^{1/3}\ (log(log\ n))^{2/3}}), x>1)$. time required on a classical computer. As a result, p and q can be recovered from a public key, and the corresponding private key can be calculated, breaking the security. Most other asymmetric encryption algorithms, such as ECDSA (Elliptic Curve Digital Signature Algorithm), were also broken, since they ultimately rely on problems that depend on the discrete logarithm problem, a problem that Shor's algorithm is capable of solving.[5,6]

Fortunately, hash functions and symmetric encryption algorithms are believed to be secure from quantum computing. The only quantum attack that can be used against these algorithms is Grover's algorithm, which finds the position of a known element in a set of size $n$ in $O(\sqrt{n})$ time. This is not very threatening, since hashes are generally very large.[7] For example, SHA-256[8] creates hashes of size $2^{256}$, meaning the time needed to crack it with Grover's algorithm is somewhere in the ballpark of $2^{128}$ iterations, which keeps it safe.

As a result of quantum computing, many other schemes for asymmetric encryption have been invented. These are currently secure because their best-known general attack is through Grover's algorithm, which still requires a large amount of time due to the square root of a large power of 2 still being a large power of 2. Areas of focus for post-quantum cryptography include hash-based cryptography, code-based cryptography, multivariate cryptography, and lattice-based cryptography.[9,10] However, all of these areas have barriers that prevent them from completely replacing existing schemes, such as lattice-based cryptography only being effective against worst-case attack complexity, not average-case.[11]

Hash-based signatures are of particular interest because their security directly relies on the security of the hash function used, meaning when a given security level is deemed insecure, the hash function itself can simply be replaced to increase the security level. Lamport created a one-time hash-based signature scheme, meaning each key can only be used to sign one hash before losing its security, that signs data by mapping each bit to one of two entries in the private key, and keeping a hash of every entry in the public key.[12] Winternitz created another one-time signature scheme with a reduced risk of the key being insecure where the public key is computed by hashing the private key 256 times. Signing a message is done by hashing it, then hashing each element of the private key $p_i$ a total of $256\text{-}b_i$ times, where $b_i$ is the corresponding byte of the input hash, and storing the result in the respective part of the signature. The signature can then be verified by hashing each hash in the signature $b_i$ times and comparing the result against the public key.[13] Compared to Lamport signatures, Winternitz signatures are smaller due to only needing to store a hash for every byte of the input instead of every bit. Finally, there is also a few-time signature scheme called Hash of Random Subsets that is similar to Lamport signatures but is constructed in a way that not all security is lost from reusing the key.[14]

In addition to these one-time and few-time signature schemes, there are hash-based signature schemes that allow many uses through the switching of keys after each signature. One significant hash-based signature scheme is the Extended Merkle Signature Scheme (XMSS), a scheme where a hash tree is maintained, ending in keys for a one-time or few-time algorithm such as Lamport or Winternitz. Each time a message needs to be signed, the first unused key in the tree is used to sign the message. This allows a single large tree to be used to sign a fixed number of messages, although it still requires the signer to maintain state to remember which key to use next. This need to maintain state was removed in SPHINCS, a scheme similar to XMSS that uses a few-time signature scheme where instead of using the first unused key for the signature, the key index is computed directly from the input data.[15] While this algorithm does have a small risk of reusing a key in the tree, a sufficiently large tree does not have this issue. However, because any path in the tree needs to be shared with anybody who needs to verify the signature, the tree depth also needs to be small enough to be able to efficiently share it.

## Methods

The obvious issue with every hash-based signature scheme is that either there is a limit to the number of times a given key can be used, or the key and/or signature is extremely large.

When one party needs to communicate a sequence of data to another party, this can be solved with a certificate-like approach, where each key signs a new public key together with the main data. This allows for a variably long series of secure messages.

To implement this, a hash-based signature scheme S must be split into a hash function H and a scheme-specific transformation T, such that given any message m, $S=T(H(m))$. For example, in RSA, T is the decryption function. In the case of Lamport, T is the process of taking each bit and locating it in the private key to obtain a signature element, while in Winternitz, T is the process of hashing each unit of data in the public key a given number of times.

The most straightforward method of replacing keys using certificates would then involve prepending the hashed data with a new public key and applying a scheme-specific transformation to the entire sequence as the signature, storing the new public key as the signer's state. The signature could then be verified by decrypting the signature, verifying the data hash, and storing the new public key.† This scheme is shown in Figure 1. Note that unlike a traditional stateful signature scheme, this scheme requires that the verifier stores the most recent public key as their own state.

Suppose there is a scheme with a scheme-specific transformation of T and an inverse of T_inv, a key-generation algorithm KeyGen, and a public key derivation algorithm PublicKey, and there is a hash function H that outputs b_h bits. Then, signing a message m with private key sk of size N uses the following pseudocode:

```
(sk_new, pk_new) = KeyGen(1^N)
secure_data = pk_new || H(m)
signature = T(sk, secure_data)
```

To verify this signature, the following pseudocode can be followed:

```
secure_data = T_inv(signature)
pk_new = secure_data[:N*b_h]
assert H(m) == secure_data[N*b_h:]
```

While this scheme looks ideal, the issue is that to sign a message with n bits, $2nb_k$ bits are needed in a Lamport private key if each bit corresponds to a $b_k$-bit value in the private key; $b_k$ will be referred to as the key range. This means $2Nb_k$ bits are needed in the public key where $b_h$ is the number of bits in the hash, which is greater than the n bits that this key is able to sign. Although this could be solved by reducing the key size over time, that brings back the issue of there being a limited number of signatures.

Instead, this can be solved by applying the scheme-specific transformation the hash of the key followed by the hash of the data and prepending the transformed result with the new public key, storing the result as the signature. The signature can then be verified by decrypting the transformed part and comparing the hashes for both the new public key and the data. This removes size issues due to the required signature size for a key now being $2b_h$, so any key size above that threshold can be used. This modified scheme is shown in Figure 2.

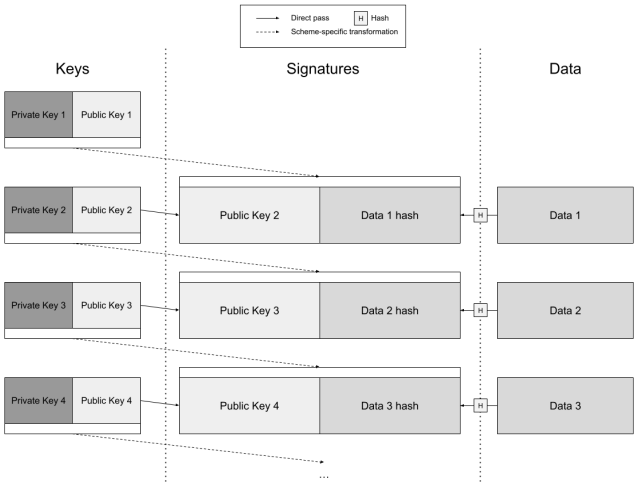


**Figure 1:** Intuitive layout of key replacement signatures

When using a linear list of keys for this algorithm, the range of numbers allowed in the key was tested for $2^{16}$ and $2^{256}$. The case of $2^{16}$ computed very fast, but the $2^{256}$ case took too much memory as a result of the code responsible for preventing random numbers from repeating themselves taking too much memory. Fortunately, for a sufficiently large range, it is extremely unlikely for a number to repeat, so after a key range threshold of $2^{17}$, this anti-repetition code was disabled. In this situation, the time complexity is $O(b_h^2+n+b_i)$ (proved in the appendix) where $n$ is the private key size, $b_h$ is the number of bits in the hash, and $b_i$ is the input size. This is similar to the Lamport time complexity of $O(n+b_i)$, and for sufficiently large key sizes, it becomes identical.

In this new scheme, signing a message m with private key sk of size N uses the following pseudocode:

```
(sk_new, pk_new) = KeyGen(1^N)
secure_data = H(pk_new) || H(m)
encrypted_secure_data = T(sk, secure_data)
signature = pk_new || encrypted_secure_data
```

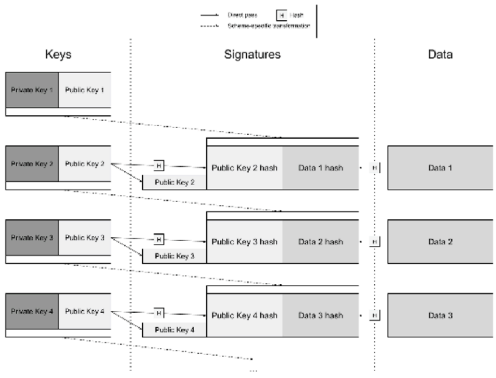


**Figure 2:** Ideal layout of key replacement signatures

To verify this signature, the following pseudocode can be followed:

```
pk_new = signature[:N*b_h]
encrypted_secure_data = signature[N*b_h:]
secure_data = T_inv(encrypted_secure_data)
assert H(pk_new) == secure_data[:b_h]
assert H(m) == secure_data[b_h:]
```


DOI: 10.36838/v5i4.14

† A similar scheme was described by Jonathan Katz,[3] but unlike that scheme, certificate-based key replacement is intended for a series of messages between two parties.

## Results and Discussion

The data collected while testing Lamport with key replacement consisted of the security level, running time, and the running time for existing algorithms at the same security level. The times are measured in seconds and were collected for both signing and verifying.

All data were collected on a 3.2 GHz Intel Core i7 processor with 32 GB 2667 MHz DDR4. The Lamport algorithm was self-implemented, while pycryptodome 3.11.0 was used for RSA, DSA (Digital Signature Algorithm), and ECDSA.[16] The software code to reproduce these results can be found at †.

First, Lamport signing with key replacement using SHA-512 (key size of 2048) was compared with RSA, DSA, and ECDSA, with random 32-byte messages. Common RSA key sizes were used with their corresponding security levels,[17] and Lamport key ranges were chosen so that the security level is approximately equal to the corresponding RSA security level, using the formula $S=log_2(b_h+2)+b_k-1$ for $b_h$ bits in the hash and $b_k$ bits per key entry, proved in the appendix. In this case, $S=log_2(514)+b_k-1=b_k+8$, which means the key entry size for a given security level in this experiment is $S$ is $b_k-8$. The data is shown in Table 1.

From Table 1, it is clear that using certificate-based key replacement with Lamport signatures takes roughly 50 times as long as DSA and ECDSA, and a varying amount with RSA. The comparison with RSA is significant because it shows that although Lamport key replacement takes longer, it is ultimately faster than what will happen with RSA, so if RSA can be made to work efficiently at a security level of 256, Lamport signatures with key replacement can as well.

The verification data in Table 2 was taken at the same time as the above signing data, using the signatures generated from the signing phase. One important change in RSA to obtain verification data that grows similarly to signing data was changing e from the typical 65537 to $2^{k-1}+1$ for key size $k$. As a result, it should be noted that in practice, RSA verification is much faster than the times indicated in Table 2.

**Table 1:** Signing times for 4 signature schemes at various security levels

| Security level | Lamport with Key Replacement | RSA | DSA | ECDSA |
|---|---|---|---|---|
| 80 | 0.11562 | 0.00081 | 0.00027 | N/A |
| 112 | 0.17746 | 0.00488 | 0.00053 | N/A |
| 128 | 0.29988 | 0.01487 | 0.00079 | 0.00082 |
| 192 | 0.35260 | 0.16313 | N/A | 0.00131 |
| 256 | 0.53875 | 0.93614 | N/A | 0.00236 |

**Table 2:** Verification times for 4 signature schemes at various security levels

| Security level | Lamport with Key Replacement | RSA | DSA | ECDSA |
|---|---|---|---|---|
| 80 | 0.03289 | 0.00031 | 0.00019 | N/A |
| 112 | 0.04754 | 0.00205 | 0.00063 | N/A |
| 128 | 0.08333 | 0.00578 | 0.00122 | 0.00159 |
| 192 | 0.09391 | 0.07413 | N/A | 0.00370 |
| 256 | 0.13701 | 0.39958 | N/A | 0.00485 |

This data shows that certificate-based key replacement with the Lamport scheme is also slower than existing algorithms, being slower than DSA and ECDSA roughly by a factor of 30. Similar to signing, RSA is unique among the schemes because Lamport with key replacement becomes faster than the RSA with a larger e at a security level of 256, and minimum security levels are bound to increase in the future.

In most cases, Lamport with key replacement is much slower than discrete logarithm-based schemes for both signing and verification. However, this slowness would be acceptable even if the scheme was not quantum-safe, and the quantum safety makes it optimal compared to the other schemes.

Compared with XMSS and SPHINCS, certificate-based key replacement has the immediate advantage of having a smaller key size. The key size is double the size needed for the underlying algorithm, whereas XMSS and SPHINCS have signature sizes that are high multiples of the base algorithm's size. Additionally, this key replacement scheme can be used infinitely, while XMSS can only be used a set number of times, and SPHINCS can only be used a finite number of times unless the tree is extremely large so that few enough input hashes map to each public key.

One disadvantage of certificate-based key replacement with single use signature schemes such as Lamport signatures is the fact that state must be updated every time data is signed, and the size of the updated state is large. This is important, since in a scenario where many parties need to verify a set of signatures, they need to verify the entire list of keys as they would with a certificate path, or there would be a risk of one of the keys being created by an attacker, and used to sign the rest of the chain of keys.

However, this can be solved by using key replacement on SPHINCS instead of a one-time scheme and updating the key every $N$ signatures using a similar certificate pattern, as shown in Figure 3. When doing this, there is a trade-off between tree size and state when a given security level is desirable. This can be seen intuitively by the fact that the most primitive version of SPHINCS would be directly passing to the onetime signature scheme, where a key update must be done after every signature, while with the largest SPHINCS key where every input is mapped uniquely, a key update is never required. Assuming the input is infinitely large, the optimal relationship between key size and signatures per state update

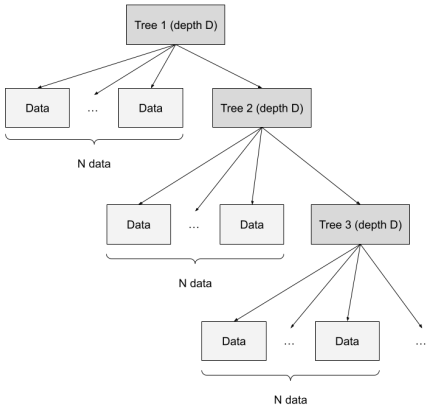


**Figure 3:** Key replacement on a many-time signature scheme

DOI: 10.36838/v5i4.14

$N = \lfloor \frac{\frac{L+L_h}{L_k+L_h}+1}{2c} + 1 \rfloor$ where given any unit of size (e.g., bits), $L$ is the total size of the tree, $L_h$ is the signature size, $L_k$ is the individual key size, and $C$ is a security parameter. The proof for this is given in the appendix.

Certificate-based key replacement is safe from both general forgery and existential forgery. It provides non-repudiation, given that the underlying algorithm used, such as Lamport, is also safe from these attacks. Because the signing of the data itself is almost identical to the underlying algorithm, there is no issue there; as a result, it suffices to show that the key replacement is secure. First, existential forgery is not a problem with the key replacement because the key is hashed, so the hash function must be reversed to get any meaningful public key, and even if this was done, more hashes would need to be reversed to get the private key or sign a message; because quantum computers cannot efficiently reverse hashes, this remains true for quantum computers. Certificate-based key replacement is also safe from general forgery because the only new channel for forgery is to use a different replaced key to legitimately sign future messages. Still, because the new key is effectively signed by the underlying algorithm, this remains safe provided that the underlying algorithm is safe. Finally, non-repudiation is provided because claiming that any data was not legitimately signed falls back on the security of the underlying algorithm, either because the key used to sign the message was broken or an earlier key was broken that led to the most recent key not being generated by the intended signer.

## Conclusion

A persistent stream of hash-based signatures with a small signature size can be achieved by replacing keys in a certificate-like manner. When applied to the Lamport signature scheme, this showed reasonable performance with the signing time growing slower than that of RSA signatures. It has the advantage of post-quantum security that is not present in traditional signature schemes. Certificate-based key replacement also has the advantage of having a smaller key size than SPHINCS. It can be used when there is a need for repeated public key signing during communication between two parties. The next step to research is the application of key replacement to many-time signature schemes such as SPHINCS, which may have useful applications such as man-in-the-middle resistant key exchange.

## Acknowledgments

I would like to thank Dr. Dong for helping me learn relevant cryptographic concepts and guiding my research. Dr. Dong was an excellent mentor for me in this project, and this paper would not exist without him. I would also like to thank Polygence for helping structure my research and for providing an excellent platform that fosters research.

## References

1. Shor, P. W. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.* **1997**, *26* (5), 1484–1509. https://doi.org/10.1137/S0097539795293172.
2. Lu, C.-Y.; Browne, D. E.; Yang, T.; Pan, J.-W. Demonstration of a Compiled Version of Shor's Quantum Factoring Algorithm Using Photonic Qubits. *Phys. Rev. Lett.* **2007**, *99* (25), 250504.
3. Katz, Jonathan. Digital Signatures. Boston, MA: Springer US, 2010. https://doi.org/10.1007/978-0-387-27712-7.
4. Lenstra, A. K. Key Lengths. *Handb. Inf. Secur.*
5. Beckman, D.; Chari, A. N.; Devabhaktuni, S.; Preskill, J. Efficient Networks for Quantum Factoring. *Phys. Rev. A* **1996**, *54* (2), 1034–1063. https://doi.org/10.1103/PhysRevA.54.1034.
6. Weisstein, E. W. *Number Field Sieve*. https://mathworld.wolfram.com/ (accessed 2022-08-21).
7. Fluhrer, S. Reassessing Grover's Algorithm. 9.
8. Technology, N. I. of S. and. Secure Hash Standard; Federal Information Processing Standard (FIPS) 180-1 (Withdrawn); U.S. Department of Commerce, 1995. https://doi.org/10.6028/ NIST. FIPS.180-1.
9. Bernstein, D. J. Introduction to Post-Quantum Cryptography. In Post-Quantum Cryptography; Bernstein, D. J., Buchmann, J., Dahmen, E., Eds.; Springer Berlin Heidelberg: Berlin, Heidelberg, 2009; pp 1–14. https://doi.org/10.1007/978-3-540-88702-7_1.
10. Ajtai, M. Generating Hard Instances of Lattice Problems (Extended Abstract). In *In Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*; ACM, 1996; pp 99 –108.
11. Peikert, C. Public-Key Cryptosystems from the Worst-Case Shortest Vector Problem. *Electron. Colloq. Comput. Complex. ECCC* 2008.
12. Lamport, L. Constructing Digital Signatures from a One Way Function. **1979**.
13. Buchmann, J.; Dahmen, E.; Ereth, S.; Hülsing, A.; Rückert, M. On the Security of the Winternitz One-Time Signature Scheme. In Progress in Cryptology – AFRICACRYPT 2011; Nitaj, A., Pointcheval, D., Eds.; Lecture Notes in Computer Science; Springer: Berlin, Heidelberg, 2011; pp 363–378. https://doi.org/10.1007/978-3-642-21969-6_23.
14. Yuan, Q.; Tibouchi, M.; Abe, M. On Subset-Resilient Hash Function Families. *Des. Codes Cryptogr.* **2022**, *90* (3), 719–758. https://doi.org/10.1007/s10623-022-01008-4..
15. Huelsing, A. *XMSS: Extended Hash-Based Signatures*. https://trac.tools.ietf.org/id/draft-irtf-cfrg-xmss-hash-based-signatures-01.html#rfc.section.4.1.9 (accessed 2022-08-03).
16. *Welcome to PyCryptodome's documentation — PyCryptodome 3.15.0 documentation*. https://pycryptodome.readthedocs.io/en/latest/index.html (accessed 2022-08-21).
17. Barker, Elaine. "Recommendation for Key Management Part 1:General." National Institute of Standards and Technology, January 2016. https://doi.org/10.6028/NIST.SP.800-57pt1r4.
18. Rachmawati, D.; Tarigan, J. T.; Ginting, A. B. C. A Comparative Study of Message Digest 5(MD5) and SHA256 Algorithm. *J. Phys. Conf. Ser.* **2018**, *978*, 012116. https://doi.org/10.1088/1742-6596/978/1/012116.

## Author

Matthew Grupenhoff is a senior at Henry M. Gunn High School in Palo Alto, CA. He intends to major in math or computer science, with a focus on cryptography and machine learning.

## Appendix

***Proof of Lamport with Certificate Key Replacement time complexity:***

Assuming the same hash function is used for both the data hash and the hashing of key entries, and this function outputs $b_h$ bits, a total of $2b_h$ public key entries are required, which results in $4b_h$ hashes being required to compute the public key. For signing and verification, only half of these are computed, resulting in $b_h$ key hashes required. In all cases, the number of hashes that need to be computed is $O(b_h)$.


DOI: 10.36838/v5i4.14

Because a hash with an $N$ bit input requires $O(N)$ time,[18] if each key entry is $b_k$ bits, then the computation of each hash requires $O(b_k)$ time, which means key generation, as well as the part of signing and verification outside the root hashing require $O(b_h b_k)$ time.

When the main hashes are included, a total of $b_k$ bits must be hashed to authenticate the new key, and $b_i$ bits must be hashed to sign the input. Because of this, the main hashing component of this signature scheme grows with $O(b_k+b_i)$ time.

Adding these time complexities, the resulting time complexity is $O(b_h b_k+b_i)$. Since the total number of bits in the key is $4b_h b_k$, this is equivalent to $O(n+b_i)$, which is identical to the standard Lamport time complexity. However, the new public key, which has a size of $b_h^2$, has not been accounted for in this equation. When that is added, the time complexity is $O(b_h^2+n+b_i)$.

***Proof of Lamport with Certificate-based Key Replacement security level:***

The main way to crack a certificate-based key replacement signature is to obtain the relevant part of a private key at any iteration. To do this, $2n$ hashes must have preimages computed for a complete private key, where $n$ is the total number of bits that can be handled by the private key (to sign both the next key and the data itself). However, if the goal is only to replace the current signature or to create a different chain of keys, then only $n/2$ entries are relevant. Of these $n/2$ entries, the needed preimage for $n/4$ of the entries, on average, will already be known after a signature is given, leaving one hash per entry for $n/4$ entries needing a preimage computed. Therefore, let $N$ be the number of hashes whose preimages are needed, which is $N/2$ if a signature is currently not known, or $N/4$ if a signature is known.

Assuming each private key entry contains $b_k$ bits and the hash contains $b_h$ bits, let $b$ be the minimum of the two; each private key entry can be any of $2^b$ values. The optimal attack would consist of comparing each hash obtained with all $N$ hashes to be reversed before checking another hash, skipping public key entries that have already been calculated; because the average number of attempts required per entry is then $2^b/N$, the total number of attempts is $\frac{N2^b+(N-1)2^b+\ldots+2^b}{N}=\frac{2^b(1+2+\ldots+N)}{N}=\frac{2^bN(N+1)}{2N}=$

$2^{b-1}(N+1)$ $2^{b-1}$ $(N+1)$ on average. As a result, the security level is

$log_2(2^{b-1})(N+1))+1=log_2$ $(N+1)+log_2 2^{b-1}+1=log_2$ $(N+1)+b-1+1$, which becomes $log_2(N+1)+b$. Assuming the scenario where this algorithm would be weakest, the security level is $log_2\binom{n}{4}+1)+b=log_2\binom{n+4}{4}+b=$

$log_2(n+4)+b-log_2 4$. This simplifies to $log_2$ $(n+4)+b-2$.

In cases where the same hashing algorithm is used for both the new key/signed data and the individual private key entries, and the key range is greater than or equal to the number of possible hashes (meaning $b=b_h$), $n=2b$ due to the number of entries for each of the two parts of the signature being equal to the number of bits. As a result, the security level becomes $log_2$ $(2b+4)+b-2=$ $log_2(2(b+2))+b-2=log_2$ $(b+2)+log_2 2+b-2$, which simplifies to $log_2(b+2)+b-1=log_2$ $(b_h+2)+b_h-1$.

If the same hashing algorithm is used, but the key range is less than or equal to the number of possible hashes, then $b=b_k$, but $n=2b_h$ still holds, so this expression becomes $log_2(2b_h+4)+b_k-2=log_2(2(b_h+2))+b_k-2=log_2(b_h+2)+b_k+log_2(2)-2$, which simplifies to $log_2(b_h+2)+b_k-1$.

***Proof of C formula:***

In many-time key replacement, $D$ and $N$ can be varied to adjust the variables. $1/N$ is the $S$ value here, while $L$ is the total size of a tree in any unit, which is $L_k$ $(2^D-1)+L_h$ $(2^D-2)$. As a result, $L=2^D(L_k+L_h)-L_k-2L_h$, so $D=$ $log_2(\frac{L+L_k+2L_h}{L_k+L_h})$.

Since there are $2^{D-1}$ unique keys in the tree, after $n$-1 unique keys are used, the probability of key $n$ being a copy of another key is $\frac{n-1}{2^{D-1}}$, and the probability of key $n$ being unique is $1-\frac{n-1}{2^{D-1}}=\frac{2^{D-1}+1-n}{2^{D-1}}$.

As a result, the probability that all keys are unique after $n$ keys $\frac{n-1}{2^{D-1}}$, are used is $\prod_{k=1}^{n}\frac{2^{D-1}+1-k}{2^{D-1}}=\frac{\prod_{k=1}^{n}(2^{D-1}-(k-1))}{2^{n(D-1)}}\geq\frac{2^{n(D-1)}-(n-1)2^{(n-1)(D-1)}}{2^{n(D-1)}}$ . As a result,

the minimum probability of all keys being unique up to that point is $1-(n-1)2^{(n-1)(D-1)-n(D-1)}=1-\frac{n-1}{2^{D-1}}$, and the maximum

probability of there being a repeated key is $\frac{n-1}{2^{D-1}}\leq\delta$, where $\delta$ is to be minimized. Defining $C=1/\delta$ (making $C$ a measure of how secure the signature is), this becomes $\frac{n-1}{2^{D-1}}\leq\frac{1}{C}$,

so $C\leq\frac{2^{D-1}}{n-1}$. For this to be true for all $n$, $C\leq\frac{2^{D-1}}{N-1}$.

In terms of $L$, this becomes $C\leq\frac{2^{log_2(\frac{L+L_k+2L_h}{L_k+L_h})-1}}{N-1}=\frac{\frac{L+L_k+2L_h}{L_k+L_h}}{2(N-1)}=\frac{1+\frac{L+L_h}{L_k+L_h}}{2(N-1)}$.

As a result, $N-1\leq\frac{1+\frac{L+L_h}{L_k+L_h}}{2C}$, and $N\leq\frac{\frac{L+L_h}{L_k+L_h}+1}{2C}+1$,

where $C$ is a given security parameter. For values of $C$ that do not yield

a fraction, the optimal value of $N$ is $N=\frac{\frac{L+L_h}{L_k+L_h}+1}{2C}+1$; the greatest integer that works when this yields a

fractional value is $N=\lfloor\frac{\frac{L+L_h}{L_k+L_h}+1}{2C}+1\rfloor$, and this also applies when the argument is an integer.

DOI: 10.36838/v5i4.14