# Time-Coordinated Signatures

Santi J. Vives
jotasapiens.com/research

**Abstract:** Hash-based signatures are typically stateful: they need to keep a state with the number of past signatures to know which values have been already used and cannot be reused. If the memory storing the state fails, the security would degrade. Some implementations solve the problem by using a number of secret values so large that the probability of picking the same at random is negligible, but this solution can make the signatures impractical for some real world applications. This paper proposes a new approach to hash-based signatures: we show that it is possible to derive their state entirely from time, without the need to keep a state with the number of past signatures.

*Keywords: many-times signatures, hash-based signatures, post-quantum cryptography, stateless, authentication, merkle tree, directed graph, time, clock.*



## 1. Introduction

Among post-quantum signatures, hash-based are the most likely to remain secure as they are based on minimal assumptions [1] [2].

Hash-based signatures consist in a series of values that are initially secret and become public as signatures are generated. Each values can only be used once, or at most a few times. As a consequence, hash-based schemes are typically stateful: they need to store in memory the number of past signatures, to know which values have been already used and cannot be reused. If the memory fails, and the recorded state is incorrect, the security would degrade.

Some implementations solve the problem by using a number of secret values so large that the probability of picking the same at random is negligible [5] [6]. But this solution can make signatures impractical for some real world applications, as they can be costly and large.

It is needed a practical signature scheme that relies on minimal assumptions, and has a security that would not break under a simple memory error. This paper proposes a new approach to hash-based signatures: we show that it is possible to derive their state entirely from time, removing the need to store the number of past signatures in memory.

Through the paper, we will take a basic signing algorithm, and make a series of modifications and improvements until we arrive to a time-coordinated signature. We will start by transforming a one-time signature (OTS) into a scheme capable of signing an unlimited number of messages. Then, we'll learn how to remove the need of storing a state, by deriving the state from time. Later, we will make optimizations to the scheme.

## 2. Chain of signatures

Let's start by transforming a one-time signature (OTS) into a scheme able to authenticate an unlimited number of messages. An OTS can only authenticate one, so we need to generate a new OTS for each message [3] [4].

We'll start by generating a private and public key pair in our first OTS. When we sign a message, we need to make sure we make available a new OTS, to authenticate our next message. So, with each message we will generate the keys of the the next OTS, and sign both the message and the new public key.

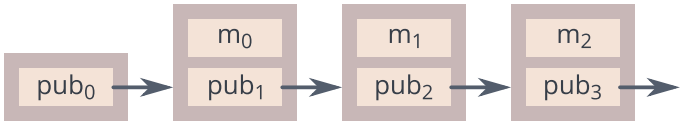


Figure 1

The result is illustrated is Figure 1: from each public key, we can prove the authenticity of a message and the next public key.

> Notice that in this paper we use the arrow $A \rightarrow B$ to mean that from a value $A$ we can prove a value $B$. A is a public key, B is the value we prove (it can be a new key), and the arrow (the proof itself) is a one-time signature.

## 3. A stateless-stateful chain

There is a problem with the chain of signatures: one needs to keep track of all the messages ever signed. Signing a new message requires a path from the initial OTS to the OTS with the message itself. That path contains all previous messages. If a memory loss occurs and the past messages become unknown or corrupted, the path would reuse OTSs differently than before, breaking its security.

To solve this, we will fork the chain in two parts: one stateless, and one stateful.

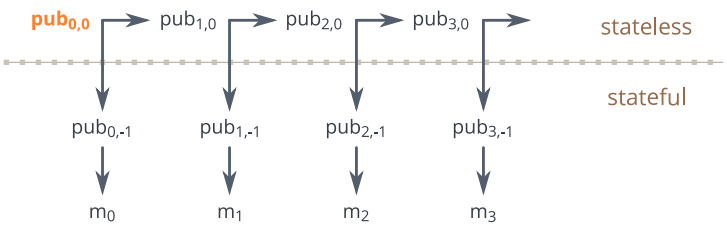


Figure 2

Using the secret key in counter mode, we will generate two sequences of keys $\{pub_{0,0}, pub_{1,0}, pub_{2,0}, ...\}$, and $\{pub_{0,-1}, pub_{1,-1}, pub_{2,-1}, ...\}$. Using the OTS corresponding to each $pub_{n,0}$ we will sign and prove both the next $pub_{n+1,\ 0}$, and the key $pub_{n,-1}$. Using the OTS from each key $pub_{n,-1}$ we will sign and prove the message itself (figure 2).

The path from $pub_{0,0}$ to $pub_{n,0}$ is now independent from the messages, and will remain the same every time it is computed (it is stateless). So is the new path from $pub_{n,0}$ to $pub_{n,-1}$. The path from $pub_{n,-1}$ to the message $m_n$ is stateful, but only depends on the message being signed. It is now possible to sign new messages without knowing the previous ones.

> OTS (one-time signature) is a bit of a misnomer. It's not exactly correct that it can only be used once: it can only be used to sign one message. Since an OTS is entirely deterministic, using it many times to sign the exact same message will not leak additional information and will not break it.

## 4. Time coordinates

We don't need to store the messages anymore, but we still need to know the *number* of messages we have signed.

To solve this, we can derive our state entirely from time: we'll assign each OTS to a coordinate in time (figure 3):

- When creating our public key, we will assign a time reference to the node with the initial key $pub_{0,0}$ (for example, *2017-jan-1 0h:0m:0s*) . And we will define a time interval (for example, 10 minutes) between pubkeys at consecutive positions (n,0), and (n+1,0). We will call these nodes in the graph time coordinate (0,0), time coordinate (n,0), time coordinate (n+1,0), and so on.
- We will collect in a buffer the messages to sign.
- We will set a scheduler at the time interval. Each time the scheduler triggers, we will combine the messages in a Merkle tree [7] and sign its root using the keys at the coordinate (n,−1) that correspond to the current time. Then we will clear the buffer for the next time.

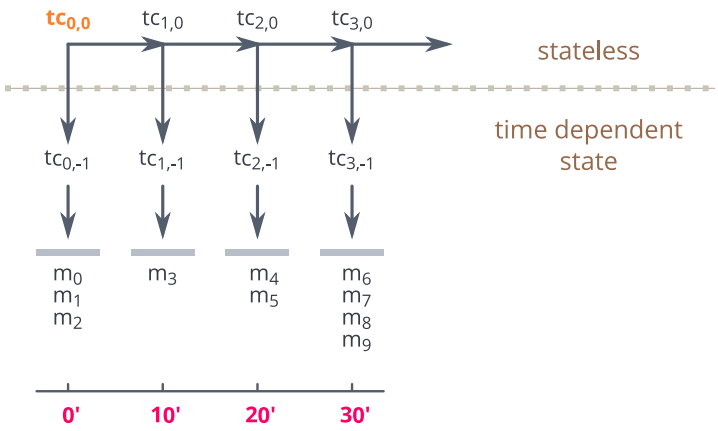


Figure 3

In our example:

- coordinates (0,0) and (0,−1) correspond to *2017-jan-1 0h:**0m**:0s*,
- coordinates (1,0) and (0,−1) correspond to *2017-jan-1 0h:**10m**:0s*,
- coordinates (2,0) and (0,−1) correspond to *2017-jan-1 0h:**20m**:0s*,
- and so on.

It follows that our signature is secure as long as the clock we use does not move backwards in time, at least not a whole time interval.

A signature is composed of:

- a stateless path from coordinate (0,0) to (n,0), and then to (n,−1),
- a time-dependent one-time signature from (n,−1) to the root of the Merkle tree,
- and a proof of membership from the message to the tree.

# 5. Time jumps

Verifying a message requires going trough every OTS in the stateless part of the chain, from the *(0,0)* coordinate to the last. It would be better if we could jump between time coordinates, skipping the intermediate steps. For that purpose, we will extend the chain into a two-dimensional graph. The added dimension will allow us to jump coordinates, moving at a faster speed.

Let's start by defining a building module, consisting of three arrows (drawn in black in figure 4):

- From time coordinate *(0,0)* in the chain, define an arrow pointing up. This arrow will serve to raise a level in the added dimension.
- From the new arrow, add another arrow extending two coordinates horizontally. This arrow will serve to move forward in time at an accelerated speed.
- From the second arrow, add the last arrow in the module, pointing down. This arrow will serve to lower a level.

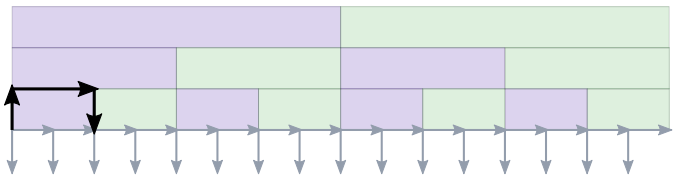

Figure 4

Now, let's repeat our module to form the entire graph. First, add infinite copies of the module, each one displaced *2* coordinates horizontally. This will create a row in the graph. Now, let's extend the graph vertically. We will make infinite copies of the row, each copy using half the horizontal resolution as the row below. Modules in the first row have an horizontal arrow that spans 2 coordinates, in the second row the arrow spans 4 coordinates, 8 in the third row, and so on. The result is illustrated in figure 5.

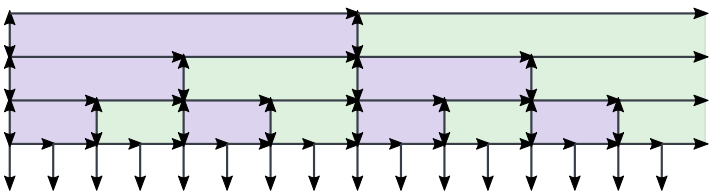

Figure 5

The horizontal arrows at higher levels permits us to move at accelerated speeds, jumping many horizontal coordinates in a single step.

# 6. Construction

## 6.1 Keys

To start, let's pick a secret seed at random, the same size in bits as the hash function output. The secret seed is the only secret value we must store: all individual private and public keys are derived from it.

At each node in the time structure there is an OTS, each with its pair of private and public keys. The arrows at the bottom of the time structure also point to other OTSs, the time-dependent ones. The privkey of each OTS can be generated from the secret seed by using a PRF and the coordinates *(x, y)* indicating its position in the time structure:

$priv_{x,y}$ = PRF *(privseed* ‖ *x* ‖ *y)*

The pubkey at each OTS can be computed from its privkey using the chosen OTS scheme.

The main public key is a value at coordinate *(0, 0)*: the node at the bottom-left in the graph. We could chose the pubkey at this node, or better, we could choose the hash with the forks at this node (see below, in 6.2). We will publish the value we choose, so that any message we sign can be authenticated using a path from coordinate *(0, 0)* to the coordinate of the message.

## 6.2 Forks

As we move in the time structure, we need to use the OTSs at each node to prove the pubkeys of all the forks we could take from there. The forks are illustrated as arrows from a node (figure 5). A node at *(x, y)* points to a series of other coordinates. We need to compute the pubkeys of all that coordinates forking from *(x, y)*, hash them, and sign the hash with the OTS at *(x, y)*. Using this method, we can prove the authenticity of any node, and move in the time structure.

From observing the graph we know that a node at coordinate *(x, y)* points to:

- $(x+2^y, y)$ if $y \geq 0$, (horizontal arrow)
- $(x, y-1)$ if $x \bmod 2^y == 0$, (down arrow)
- $(x, y+1)$ if $x \bmod 2^{y+1} == 0$, (up arrow)

The pukeys at this coordinates are the values that need to be hashed, and signed with the OTS at *(x, y)*.

## 6.3 Salted hashes

To improve security, the hash functions used in each OTS must be unique. This is done so to prevent multi-target attacks, and rely on preimage resistance, rather than collision resistance. To make each hash function unique, we will feed each one with a distinct salt value:

$H_n$ *(*$salt_n$ ‖ *input)*

#### OTS salts

Each salt in an OTS will be derived from a salt seed, generated at random and made public along with the main public key. The salt seed is made unique by appending the coordinates corresponding to the OTS:

$H_{x,y}$ *(salt_seed* || *x* || *y* || *input)*
$salt_{x,y}$ = *salt_seed* || *x* || *y*

In addition, the OTS scheme can append other indices to differentiate the hash function within the OTS.

#### Fork salts

An OTS at a node in the time structure signs all the possible forks in the path from that position. The possible forks are combined in a single hash, and then the hash is signed. That hash is salted as well. To use a different salt than the one the OTS uses, we will flip a bit in the seed:

$H_{x,y}$ *(salt_seed* ⊗ *1* || *x* || *y* || *forks)*

**Message salts**

To eliminate the possibility of signing a collision of two distinct messages *m* and *m'* generated by an attacker, we will hash the message with a salt, unknown for all but the signer. Unlike the others, this salt must be kept secret until the signature becomes public. The public key of the OTS at coordinate (−1,n) that we will use to sign the message itself satisfies this condition. Then, we can feed this pubkey as the salt:

*H (ots_pubkey || message)*

# 7. Security

The graph has two parts: one stateless, and one time-dependent.

The stateless part is composed of many OTSs, each with an unique salt, and an independent key. Each OTS always authenticates the same message. It is secure as long as it meets the following conditions:

- The scheme uses a secure OTS with safe parameters.
- Breaking one of many independent OTS is no more feasible than breaking a particular OTS.
- The salt value is set to a number large enough that a collision is unlikely.
- The secret keys are derived using a PRF.

In the time-dependent part of the scheme, each independent OTS is assigned to a time coordinate. Since each OTS is secure as long as it is guaranteed to be used only once, we have an additional security condition:

- The clock does not move backwards in time, at least not an entire time interval.

# 8. Graph optimizations

The make paths shorter, the time structure can be modified so that movements that are likely to occur together are combined in a single step.

## 8.1 Modules

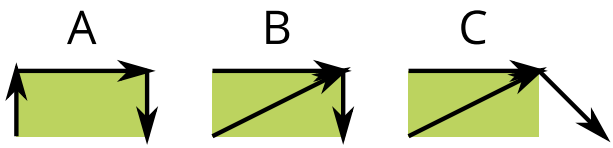


Figure 6

If the path moves up, it will later move forward horizontally. To shorten the path, the two movements can be merged in a single step. The original module (A), and the modified module (B) are illustrated in figure 6. The resulting arrow now raises a level, and moves forward at an accelerated speed. We get an acceleration arrow from a coordinate *(x, y)* that points to $(x+2^{y+1}, y+1)$.

Similarly, we can merge the arrow moving down with a forward arrow in the level below. The result is module C in the figure. We get a deceleration arrow, from a coordinate *(x, y)* to $(x+2^{y-1}, y-1)$.

Repeating the modules A, B, and C, we get the time structures A, B, and C, illustrated from top to bottom in figure 7.

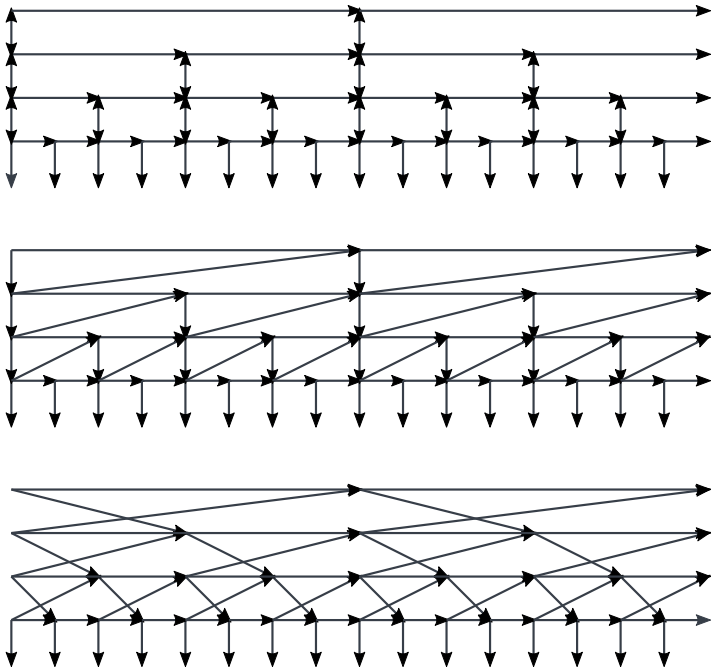

Figure 7: structures A, B, and C
(from top to bottom)

Figure 8 shows a path in each structure, going from coordinate *(0, 0)* to *(15, 5)*.

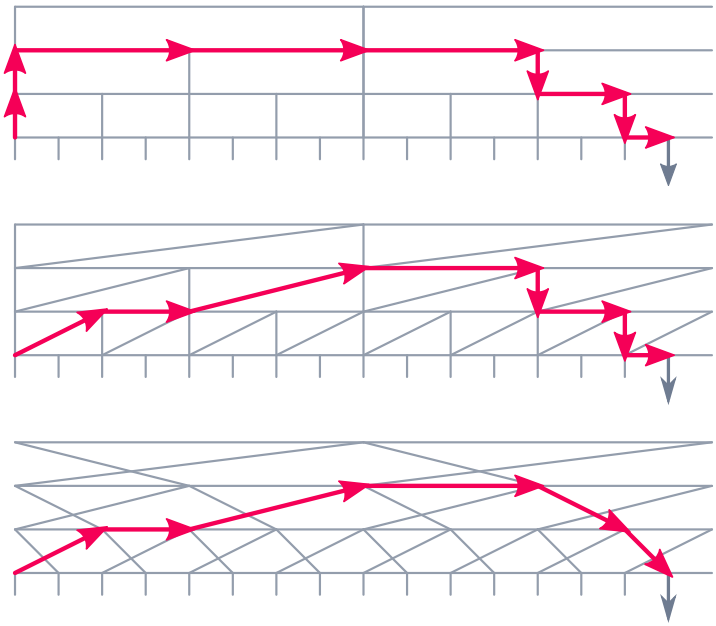

Figure 8

Each new optimization takes fewer steps: in structure A, the path takes 9 steps; it takes 8 steps in structure B; and finally, 6 steps in structure C.

### 8.2 Offset

Let's optimize the path from the *(0, 0)* coordinate. It would be better if we could start from *(0, 0)* and always move up a level in a single step (figure 9). To achieve that, after raising each level we should always meet and acceleration arrow. The coordinate positions should be shifted 2, 4, 8 steps, and so on. Then a level $n$ should begin at position $2^{n+1}-2$ relative to level zero.

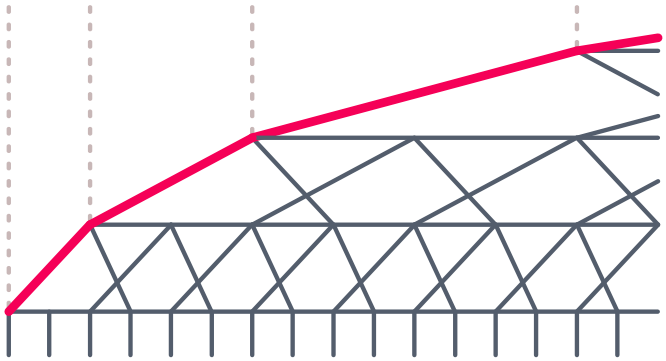

Figure 9

### 8.3 Resulting time structures

The time structures can be generated by starting at coordinate *(0,0), and extending the graph. In each time structure, a node at coordinate* (x, y)_ points to:

#### Time structure A

- $(x+2^y, y)$ if $y \geq 0$ (forward arrow)
- $(x, y+1)$ if $x \bmod 2^{y+1} == 0$ (up arrow)
- $(x, y-1)$ if $x \bmod 2^{y} == 0$ (down arrow)

#### Time structure B

- $(x+2^y, y)$ if $y \geq 0$ (forward)
- $(x+2^{y+1}, y+1)$ if $x \bmod 2^{y+1} == 0$ (acceleration)
- $(x, y-1)$ if $x \bmod 2^{y} == 0$ (down)

#### Time structure C

- $(x+2^y, y)$ if $y \geq 0$ (forward)
- $(x+2^{y+1}, y+1)$ if $x \bmod 2^{y+1} == 0$ (acceleration)
- $(x+2^{y-1}, y-1)$ if $x \bmod 2^{y} == 0$ (deceleration)

#### Time structure C with corrected offset

After subtracting the offset $2^{y+1}-2$, we get:

- $(x+2^y, y)$ if $y \geq 0$ (forward)
- $(x+2^{y+1}, y+1)$ if $(x-2^{y+1}+2) \bmod 2^{y+1} == 0$ (acceleration)
- $(x+2^{y-1}, y-1)$ if $(x-2^{y+1}+2) \bmod 2^{y} == 0$ (deceleration)

# 9. Transversal

Now we have constructed the time structure, we need an algorithm to find a path between coordinates. To find the shortest path, we can follow these simple rules:

1. We will place two points A and B, one at the first coordinate, and one at the last coordinate. We will move from both locations until the points meet in the middle.
2. We will raise a level as soon as we can, as long as moving up does not prevent us from meeting the other point.

The second rule can be rewritten as follows:

- If there is an arrow pointing up from the current position, we move up.
- If not, we move horizontally until the first node with an arrow pointing up.

> There is one exception to rule 2:
> In structure A, moving two steps horizontally is faster than raising a level, moving one step horizontally, and lowering a level again. In that case (only 1 horizontal step), we won't move up.

## 9.1 Code

The following Python 2.7 functions follows the above rules to find the shortest path in each structure. Each function counts the number of steps between two points A and B, representing two coordinates.

**Example**

**Auxiliary functions**

Two auxiliary functions, needed by the functions that compute the path.

**round_up:** round an integer $n$ up to the nearest integer of the form $g \cdot m + offset$, where $g$ and $offset$ are constants.

```
def round_up (n, g, offset=0):
        r = n - offset
        r = ((r + g - 1) // g) * g
        r += offset
        return r
```

**round_down:** Round an integer $n$ down to the nearest integer of the form $g \cdot m + offset$, where $g$ and $offset$ are constants.

```
def round_down (n, g, offset=0):
        r = n - offset
        r = (r // g) * g
        r += offset
        return r
```

**Structure A**

```
def structA (start, end):
        g = 2
        A, B = start, end
        movs = 0
        for n in count ():
                g = 2 ** (n)
                gn = 2 ** (n + 1)
                # move point A
                A_next = min (round_up (A, gn), B)
                movs += (A_next - A) // g
                A = A_next
                # move point B
                B_next = max (round_down (B, gn), A)
                movs += (B - B_next) // g
                B = B_next
                # check distance
                if B - A == gn:
                        movs += 2
                        A = B
                # end or raise level
                if B == A:
                        break
                else:
                        movs += 2
        return movs
```

**Structure B**

```
def structB (start, end):
        g = 2
        A, B = start, end
        movs = 0
        for n in count ():
                g = 2 ** (n)
                gn = 2 ** (n + 1)
                # move point A
                A_next = min (round_up (A, gn), B)
                movs += (A_next - A) // g
                A = A_next
                # move point B
                B_next = max (round_down (B, gn), A)
                movs += (B - B_next) // g
                B = B_next
                # check distance
                if B - A == gn + g:
                        movs += 2
                        A = B
                # end or raise level
                if B == A:
                        break
                else:
                        A += gn
                        movs += 2
        return movs
```

**Structure C**

```
def structC (start, end):
        g = 2
        A, B = start, end
        movs = 0
        for n in count ():
                g = 2 ** (n)
                gn = 2 ** (n + 1)
                # move point A
                A_next = min (round_up (A, gn), B)
                movs += (A_next - A) // g
                A = A_next
                # move point B
                B_next = max (round_down (B, gn, offset=1), A)
                movs += (B - B_next) // g
                B = B_next
                # end or raise level
                if B == A:
                        break
                else:
                        A += gn
                        B -= g
                        movs += 2
        return movs
```

**Corrected Offset**

Each time one raises a level the horizontal position must be shifted to correct the offset. Compared to the *structC* function, two lines are added at the end:

```
        ...
                else:
                        A += gn
                        B -= g
                        movs += 2
                        A -= gn
                        B -= gn
        return movs
```

## 9.2 Result

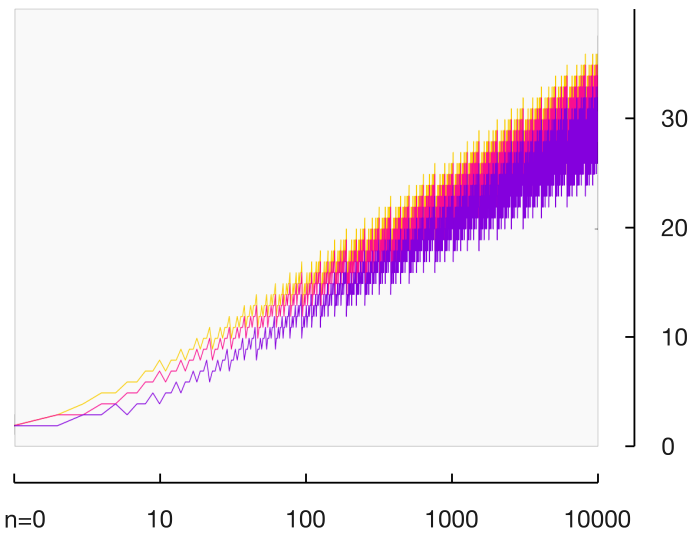


Figure 10

Figure 10 plots the length of the paths from coordinate *(0, 0)* to *(0, n)*, while moving in structures A (amber), B (pink), and C (purple).

With the exception of very small *n*, in all structures a linear increase in the path corresponds to an exponential increase in the distance between coordinates. Structure C is more efficient than structure B, and structure B is more efficient than structure A.

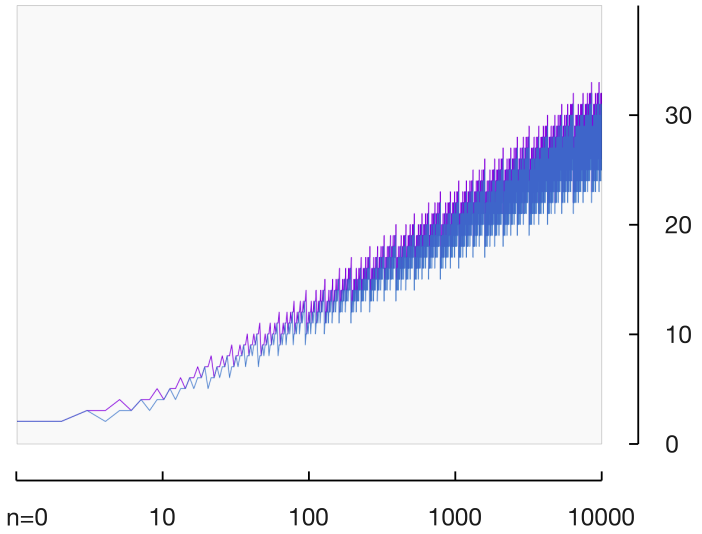


Figure 11

Figure # plots the length of the shortest paths moving on structure C (purple), and structure C with optimized offset (blue). We can see that adjusting the offset improves the structure a bit further.

# 10. Further improvements

## 10.1 Software and firmware updates

Software and firmware updates are among the applications most in need of post-quantum cryptography (that is, algorithms that run in a classical computer but are quantum-resistant). There is an important, and increasing number of devices of all kinds running software. And it's not uncommon for devices to keep running for many years (some even decades) from the time of their manufacturing. It is important then that their security is able to survive a breakthrough in quantum computing, or a mathematical breakthrough that could break classical algorithms (such as semiprimes factoring, or the discrete logarithm).

Among post-quantum algorithms, hash-based cryptography is the safest bet:

- it's based on minimal assumptions,
- it is a field that has been studied since the late 70's,
- its underlying security principles are well understood,
- and more important, an attack against hash functions (and in consequence hash-based cryptography) would also break any other form of digital signature (classical or post-quantum), while the opposite is not true: an attack against other problems (such as the discrete logarithm, or lattice problems) does not translate to an attack against hash-based signatures.

In addition, software are firmware updates have characteristics that allows a simple application of time-coordinated signatures. Version of software are naturally ordered in time: rather than relying on a clock, it is sufficient to directly assign each version (1, 2, 3, ...) to a time coordinate. Then, version 0 corresponds to coordinate 0, version 1 to coordinate 1, and so on. It follows that the scheme is secure as long as each version number is only used one: if a file software update package is signed as version *n*, that version number cannot be reused, and the next version must be *n+1* or higher.

**Efficient updates**

In this subsection we will learn how to authenticate software and firmware updates using signatures that are short and efficient.

As an example scenario, we have devices running version 36 or older, and we want to update them to version 37. Table 1 shows the amount of devices running each version. The majority of devices (61%) is currently running versions 34, 35, and 36. While a minority of devices (39%) hasn't update for a while, and is running version 33 or older.

| Version | Devices |
|---|---|
| 36 | 21% |
| 35 | 28% |
| 34 | 12% |
| 33 | 8% |
| 32 | 3% |
| 31 | 2% |
| 0 to 30 | 26% |

Table 1

Each devices stores a public key to verify future updates: it stores the initial coordinate *(0,0)*, and the last coordinate it knows. For example, a device running version 35 stores coordinates *(0,0)*, and *(0,35)*.

To update, we want to generate a signature that can be verified by all devices, and is as efficient as possible for most of them. To do so, we'll generate a path, jumping from coordinate *(0,0)* to *(0,34)*. Then, we'll extend the path from *(0,34)* to *(0,37)* without jumping. The full signature is illustrated in figure 12.

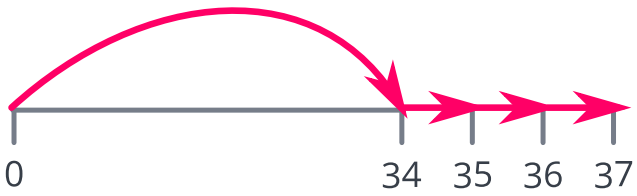


Figure 12

When a device in the majority (running version 34 or later) asks for an update, we only need to send the path from its current version to the last coordinate *(0,37)*. Only the minority of devices running version 33 or older needs the full path from coordinate *(0,0)*.

The majority of devices is able to verify the update using a very short path, with only 3 steps or less. This short signature results in a reduced bandwidth for the distribution. And in a verification with fewer operations, which translates to less energy consumption for the devices.

## 10.2 Merkle trees

A Merkle tree, and a time structure are graphs with some interesting similarities and differences:

- A Merkle tree has a finite number of nodes (leaves) at the bottom level. Each level above has half the number of nodes as the level below, until there is only one at the top (the root). The time structure has an infinite number of nodes at the bottom. Each level above has an infinite number of nodes as well, but with half the horizontal resolution. Moving up, there is an infinite number of levels.
- In a Merkle tree, a path goes from the node at the top (the root) to a any node at the bottom. In a time structure, a path goes between any two nodes at the zeroth level, provided that the first node is before the second.
- In a Merkle tree, each step in the path can be proved with a simple hash value. In a time structure, proving the path requires OTSs.

| **Merkle tree** | **Time structure** |
|---|---|
| Finite number of signatures | Unlimited signatures |
| Constant path length | Shorter path between near nodes |
| Path proved using simple hashes | Path proved using OTSs |

Table 2

Most notably, a step in the path in a Merkle tree consists of a single hash, making the step more efficient. In this subsection, we'll see how a Merkle tree, and a time structure can be used together to improve the scheme.

**Tree of forks**

In the time structure, at each node there is a hash of all possible forks that the path can take from there. That hash is signed using the OTS at the same node (except for coordinate 0,0 where the hash is sufficient).

So far, we have considered a fork with exactly 1 step in each possible direction. Instead, the hash can contain at most $k$ steps in each directions. To do so, we will compute all pubkeys at all the positions we can go to, after moving at most $k$ step in each direction (accelerating, moving forward, and decelerating). Then, we will combine the pubkeys in a Merkle tree. The root of the tree is now our hash.

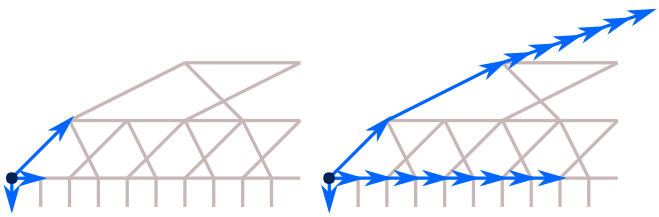

Figure 13

Figure 13 on the left shows the positions after moving exactly 1 step from coordinate (0, 0). On the right, it shows the positions after moving at most 8 steps in each direction.

A tree of forks makes it possible to move faster in the time structure, but the signer needs to compute the public keys located at more nodes. It results in a trade-off between the efficiency of verification and the cost of signing.

#### Optimization

When moving many nodes in a single step, two simple optimization become possible:

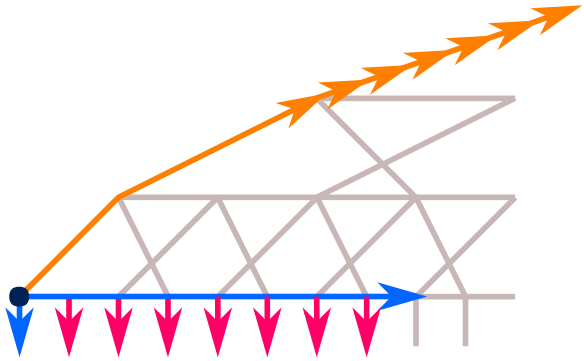

Figure 14

- **Extension:** Some times, from some nodes, there is an only possible movement. Then it would be convenient to extent the path in that only possible direction. An example is illustrated in figure 14, in pink. If one moves horizontally and stops before reaching the blue arrow, there is only one next possible move: down.
- **Removal:** Some movements are redundant, and can be omitted. For example, accelerating only one node (in yellow) only makes sense if one wants to move to a node in pink. Then, the arrow pointing only one step up can be safely omitted.

### Tree of time-structures

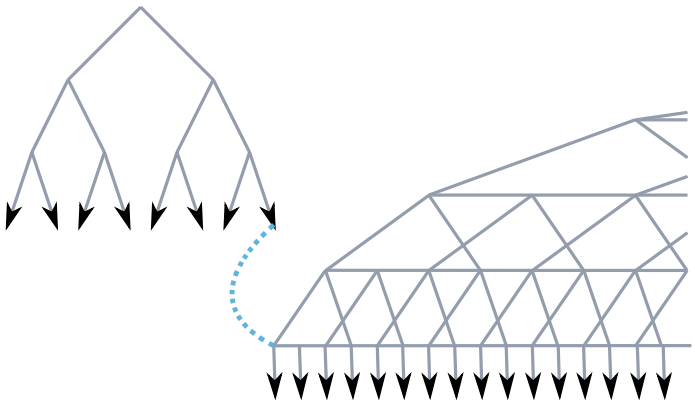

Figure 15

Rather than a single time structure, it is possible to generate many, and authenticate them using a Merkle tree. The coordinates *(0,0)* of all time structures are combined in a tree, and the root of the tree becomes the general public key.

A signature is authenticated by proving a path from the root to coordinate *(0,0)* in the time structure at a leaf, and a path from *(0,0)* to the coordinate with the message.

Then, for a tree with $f$ leaves, a coordinate $(0, x)$ is mapped to:

- a leaf $x \bmod f$ in the tree,
- and a coordinate $(0, \text{floor } (x/f))$ in the time structure.

#### Jumping between timelines

When using a Merkle tree, time coordinates are separated in different timelines. Each timeline stars at a different leaf. We want to make it possible to jump from one coordinate to another, even if they are in different timelines. We can achieve this as follows:

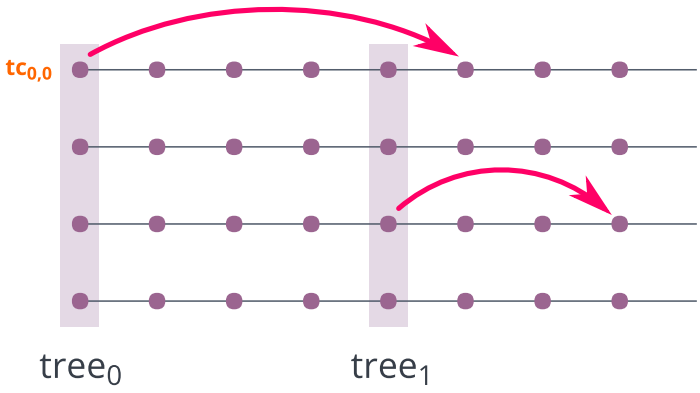


Figure 16

At a given time interval, we will compute a new Merkle tree from the public keys $tc_{0,n}$ in each timeline (figure 16). With every new signature, we append the root of the new tree to the message, and sign them together. It is now possible to jump between timelines using the latest known tree.

## 10.3 Few times signatures

Using a few times signatures instead of an OTS result in these simple improvements:

#### Path

While using an OTS to prove a path in the time structure, it is necessary to precompute and sign all possible forks from the current point. Using a few-times signature, it would be safe to sign only the path one is taking. This will speed up signing.

#### Messages (time resolution)

The OTS used to authenticate the messages can be replaced with a few-times signature (FTS). An FTS would allow to generate multiple signatures from a single time coordinate, thus increasing the time resolution.

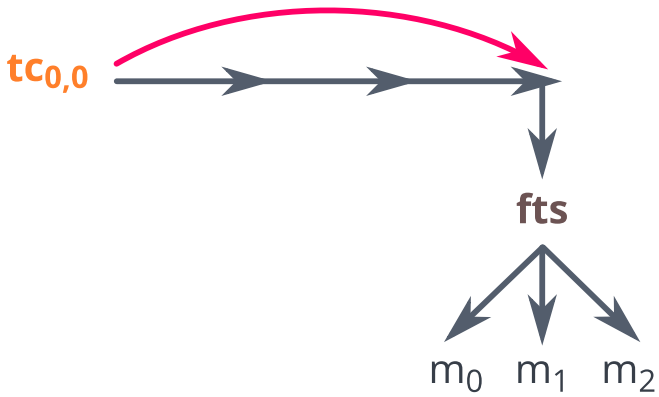


Figure 17

### 10.4 Higher dimensions

We started with a chain of one-time signatures forming a graph in a single dimension. Then we learned how to add a second dimension to the graph, useful to move at an accelerated speed. Using the same method, we could add a third dimension to move faster in the second dimension. The method can be repeated to extend the graph to an arbitrary number of dimensions.

### 10.5 Clock state

We've learned that hash based signatures, instead of relying on a memory state not failing, can rely on a clock not moving backwards. We can do better than that: we can make the possibility of a failure even more unlikely, in a way that would require both a clock and a state to fail simultaneously, and that they fail in such a way that the two errors are in agreement.

Consider this: in addition of using a clock, we'll store a state with the last time coordinate we've used. Every time the clock triggers, we will assign the current time coordinate as usual. But we will sign only if the current time coordinate is in the future of the coordinate recorded in the state.

## 11. Conclusions

We've learn how to build hash-based, many-times signatures with the following advantages:

- No need to keep a state. The scheme is secure as long as it uses a clock that does not move backwards in time.
- No need to know the maximum number of signatures in advance. The scheme permits an unlimited number of signatures per key.
- The size grows only linearly as the number of signatures grows exponentially.
- The sizes grow as more signatures are generated, but only if the verifier has never observed a signature before. In cases when the verifier already knows a time coordinate, it is sufficient to prove a path from that coordinate rather than *(0, 0)*, resulting in a shorter path.

## References


1. Dods, Smart, Stam - Hash-based digital signature schemes - chapter in Cryptography and coding (2005).
2. Buchmann, Dahmen, Szydlo - Hash-based digital signature schemes - chapter in Post-quantum cryptography (2009).
3. Buchman, Dahmen, Sarah Ereth, Hülsing, Rückert - On the security of the Winternitz one-time signature scheme (2011),
4. Leslie Lamport - Constructing digital signatures from a one way function (1979).
5. Oded Goldreich - Foundations of cryptography, volume 2, basic applications (2004).
6. Bernstein, Hopwood, Hülsing, Lange, Niederhagen, Papachristodoulou, Schwabe, Wilcox - SPHINCS: practical stateless hash-based signatures (2015).
7. Ralph C. Merkle - Secrecy, authentication, and public key systems (1979) - A certified digital signature (1989).