From One to Many: Synced Hash-Based Signatures

Santi J. Vives Maccallini
santi@jotasapiens.com

jotasapiens.com/research

Abstract: Hash-based signatures use a one-time signature (OTS) as its main building block, and transform it into a many-times scheme, to sign a larger number of signatures. In known constructions, the cost and the size of each signature increases as the number of needed signatures grows. In real-world applications, requiring a significant number of signatures, the signatures can get quite large. As a result, it is usually believed that post-quantum signatures based on hashes need more computation and much larger sizes than classical signatures. We introduce a construction to challenge that idea: we show that it is possible to construct a many-times signatures scheme that is more efficient than the OTS it is built from, rather than less. We study the generation of signatures in conjunction with a blockchain, like bitcoin. The proposed scheme permits an unlimited number of signatures. The size of each signatures is constant and the same as in the OTS. The verification cost starts the same as in the OTS and decreases with each new signature, becoming more efficient on average as the number of signatures grows.

Keywords: many-time signatures, hash, post-quantum cryptography, authentication, blockchain, bitcoin, optimization.

1. Introduction

Hash-based schemes use a one-time signature (OTS) as its main building block, only useful to sing one message per key. An OTS is then transformed into a many-times schemes, useful to sign an unlimited (or at least large) number of messages. In both, the difficulty of breaking a signature reduces to the problem of breaking the hash function they are made of.

In known many-times constructions, the cost and size of each signature grow as the number of needed signatures grows: if a key is used to sign a million signatures, those signatures will be more costly and much larger than others from a key that only signs ten. In applications that require a very large number of signatures, they can be impractical.

In this paper we will study the generation of signatures in conjunction with a blockchain, like bitcoin. A blockchain is a public, decentralized, and unforgeable database. Once inserted in a blockchain, signatures become known to anyone. We will show that when all signatures are known, it is possible to construct many-times signatures more efficient than the OTS they are built from, rather than less.

We propose a method to transform a one-time signature into an unlimited many-times scheme. In the new scheme, the size of each signature remains constant and the verification cost decreases as the number of signatures grows.

2. Background

2.1 One-Time Signatures

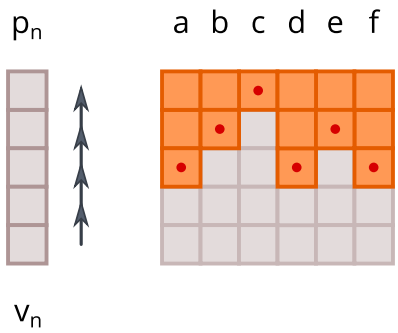
Lamport proved a method to transform a hash function into a signature that is secure as long as each key is used only once [1].

The one-time signature (OTS) does not rely on the hardness of some arithmetic problem like the discrete logarithm, nor any other assumption. Breaking the OTS reduces to the problem of breaking the hash function it is build from. So, an OTS is secure as long as the hash function is secure.

> In practice, a *keyed* hash function is used to make each hash instance unique (for example an HMAC or a salted hash). This prevents an attacker from targeting multiple hashes at once.

There are many different one-time signature schemes. We will consider a broad family of signatures that includes Lamport [1], Winternitz [2], and Integer Compositions [3] as special cases.

The OTS we will work with uses a chain as its main building block (figure 1, left). To construct a chain, a hash function is applied a number of times over a value v_n in the private key, until obtaining a value p_n in the public key. A signature requires multiple chains, represented by each column in the illustration to the right.



(Figure 1)

Generally, the it works as follows:

First, a private key is created. The key is an *L*-tuple, with each part chosen at random, with the same size as the hash function.

Then, the signer iterates the keyed hash function (for example an HMAC or a salted hash) *z* times over each value in the private key. The result is the public key.

To sign a message, a hash *m* of the message is mapped to a tuple *u*. We will call *u* the verification tuple. Each part in the tuple takes values in the range *0...z*.

A complementary signing tuple *y* is created, such that *y* and *u* add up to the *L*-tuple *(z, z, ...)*. With each value $y_n = z - u_n$.

The signer iterates the hash function over each value in the private key, a number of times given by *y*. And publishes the result as the signature. The signature is represented by the dotted blocks in the illustration.

Then, the verifier completes the path, iterating the hash function u times over each value in the signature, and compares the result against the public key.

In our example (figure 1), we have $L = 6$ and $z = 4$. The u tuple is (2, 1, 0, 2, 1, 2). And its y tuple is (2, 3, 4, 2, 3, 2). Where:

(2, 1, 0, 2, 1, 2) +
(2, 3, 4, 2, 3, 2) =
(4, 4, 4, 4, 4, 4)

Security

Once a signature becomes public, all hashes going from the signature to the public key become public as well. They can be easily computed by applying the hash function.

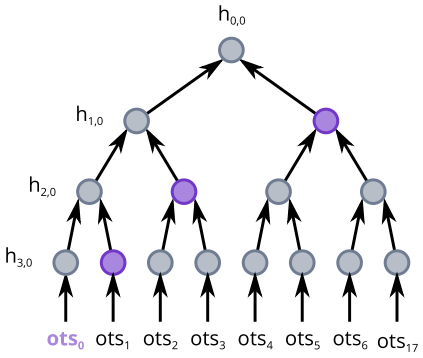
For a OTS scheme to be secure against forgery, it means that those hashes and their corresponding values in the u tuple must be insufficient to generate another signature. That condition restricts the set of tuples that can be valid within a scheme:

Applying the hash function allows to compute hashes that correspond to smaller u_n values, and their complementary larger y_n values. Then, in every possible pair of valid u tuples, each tuple must have at least one part u_n that is greater than the same part in the other tuple.

In other words:

> If a verification tuple u corresponds to a valid signature, a second, distinct tuple u' with each $u'_n \leq u_n$ does not **(lemma 1)**.
> For example, if the tuple *(0, 1, 2, 3)* corresponds to a valid signature within a scheme, the tuple *(0, 0, 2, 3)* does not.

2.2 Many-Times Signatures



(Figure 2)

Merkle proved that a one-time signature can be transformed into a many-times signature scheme [4]. To do so, he introduced the Merkle Tree (figure 2).

The signer generates as many OTSs as messages we wishes to sign. The public keys of the OTSs are hashed and placed at the leaves of the tree. Then, all values are hashed in pairs until reaching the root of the tree at the top. The root is the many-times public key.

To sign a message, the signer chooses an unused OTS and makes public the hashes needed to authenticate the OTS to the root (the purple values in the figure).

There are later variants of the scheme (CMSS, GMSS, XMSS, and more). They are out of the scope of this paper, so they'll be commented briefly. The variants replace some of the hashes in the authentication path with OTSs, so that there is no longer need to compute the entire tree when creating the keys. But, replacing a simple hash with a OTS increases the size of each signature and its verification cost.

The Problem

The Merkle Tree itself has proven to be useful in lots of applications. But it never reached widespread use as a signature scheme.

In the Merkle signature and its variants, the size and cost depend on the number of signatures. As the number grows, the tree becomes larger. And, as a result, the size and cost of each signature increase as well. The signatures can get quite large, to the point of being impractical for many applications.

In addition, the signature is stateful. The signer needs to keep track of the number of messages he signs to avoid using an OTS more than once. A memory failure can make the signer reuse an OTS and degrade the security.

2.3 Blockchain

Usually, the security of transactions relies on the assumption that a third party acting as a middleman (for example, a bank) is honest .

Bitcoin [5] removes the need for a middleman, shifting from trust in persons and institutions to trust in mathematical knowledge. Rather than relying in people not misbehaving, the authenticity and correctest of the transactions are secured using a cryptographic construction.

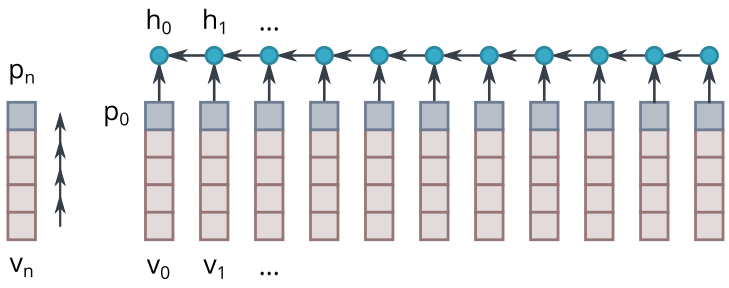
To do so, the blockchain (or chain of blocks) is introduced: it's a timestamped, decentralized, and unforgeable database. Each block in the chain contains a set of signed messages, usually combined in a Merkle Tree.

A blockchain, as a cryptographic primitive, ensures that anyone observing a block is observing the same information, without modifications nor omissions. Once included in the blockchain, anyone can observe the number of signatures generated from each key, and what those signatures are.

3. A Simple Synced Signature

In this section we will learn the basics of how synced signatures work. First, we will create the chains. Then, we will learn how to use those chains to generate synced signatures using a blockchain.

3.1 Stream



(Figure 4)

To sign many messages, rather than creating a new set of chains for each signature, we want one large set of many chains. We will take a few as needed for each signature and prove they are members of the set.

We will create a long but finite set of chains, as in figure 4, and hash each public key at the top, from right to left. Each hash value h_n takes as input the public key below it (p_n) and the hash value to the right (h_{n+1}). Will make public the leftmost hash h_0.

To prove a sequence of public keys p_0, p_1, ..., p_n, we need to publish the hash value h_{n+1}. To verify the values, they must be hashed from right to left, in the same way than when we created the stream, and the resulting value must be checked against h_0.

Extension

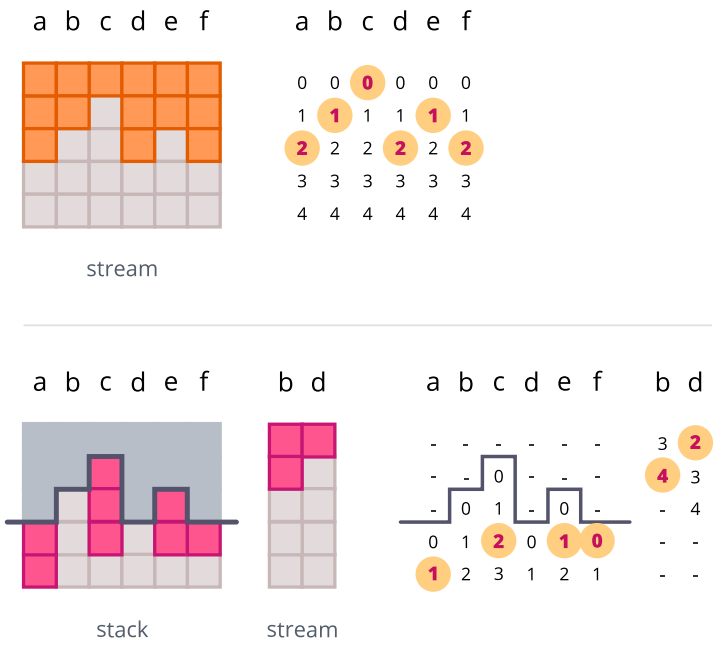
If we want an unlimited number of signatures, we should be able to extend the stream as needed. To do that, we'll simply generate a new stream, and use the previous one to sign a message together with the leftmost h'_0 value of the new stream:

sign (message || h'_0)

3.2 Signing

Let's start by generating the first signature, the same way as we did before with the OTS (figure 3, top). First, we map our message to its corresponding *u* tuple. Then, we compute the corresponding *y* tuple. Using the first chains from the stream, we iterate the hash function *y* times over the private key to compute the signature.

Figure 3, at the top, shows our first signature. In our example, *u = (2, 1, 0, 2, 1, 2)*. The illustration, to the left, shows the hashes that become public. To the right, the values from *u* that the chains encode.



(Figure 3)

We publish the signature and the h_n value to the blockchain. Now is when things start to get a bit different.

For a one-time signature to remain secure, it is important that the hashes from one signature are never reused for another. This is usually achieved by discarding the chains from an OTS after its use.

But once the signature is inserted in a public blockchain something interesting happens. The hashes going from the signature to the public key are known to everyone now, and cannot be reused. The hashes at the bottom, from the private key to the preimage of the signature, remain unknown. Since everyone can observe in the blockchain which hashes are already used, we can safely use the remaining unspent hashes to authenticate our next signature.

Now, we will generate our second signature, using the set of chains from the past signature (we'll call that set the stack), and new chains from the stream.

We will assign the possible values in u_n to the unspent hashes in each chain from the stack, from top to bottom: 0, 1, ... As shown in figure 3 at the bottom.

We will public the hashes that correspond to the *u* tuple from our second message. In our example, *u = (1, 4, 2, 2, 1, 0)*.

Since the number of unspent hashes in the stack is smaller than originally, some parts from *u* cannot be encoded by them. When a chain from the stack is insufficient to encode a part in *u*, we will take a new chain from the stream to encode the remaining values. From top to bottom, counting from where we left.

In our example, the unspent hashes in the stack can encode the following ranges of values:

a → 0...1, b → 0...2, c → 0...3,
d → 0...1, e → 0...2, f → 0...1

We will publish the hashes in the stack that correspond to values from our *u* tuple. We are able to encode four values from *u*:

u: (1, ∅, 2, ∅, 1, 0)

Two positions are missing (*b* and *d*). So, we need two extra chains from the stream. We will assign the new chains to the missing positions (letters) in order: the first chain to *b*, the second to *d*.

The chains from the stack at those positions encoded the values *0, 1, 2* and *0, 1* respectively. So, we'll assign the remaining values to the stream: *3, 4* to the first chain and *2, 3, 4* to the other. From top to bottom.

We will append to the signature the hashes that correpond to the two remaining positions in *u* :

u: (∅, 4, ∅, 2, ∅, ∅)

Figure 3, at the bottom, show our second signature. To the left, the hashes that become public with our signature. To the right, the values from *u* encoded by the the stack and the the stream.

The resulting synced signature can be represented using three tuples: u_{stack}, u_{stream}, and *pos*:

- The first tuple u_{stack} indicates the number of hash iterations to the first hashes in the signature, until obtaining the known values in the stack.
- The tuple u_{stream} indicates the number of hash iterations to last hashes in the signature, until obtaining the values at the top of each chain in the stream.
- And *pos* indicates the positions of the used the chains in the stack.

In our example, the mapping from *u* to the synced signature is:

u = (1, 4, 2, 2, 1, 0) ⟶
u_{stack}, u_{stream}, pos = (2, 3, 2, 1), (1, 0), (0, 2, 4, 5)

> Notice in the figure that in the synced signature, the 0 (zero) value of the *u* tuple isn't encoded by the upper known hash. It is encoded by the hash below it instead (its preimage). We want the signer to prove in all cases he knows a new hash from the chain.
> When using the stream, that isn't necessary, since the hash at the top is unknown.

Size

The size of the signature is the same as before. But the size of public key has expanded. Since the new signature authenticates in part to the previous signature, that previous signature has become part of the public key.

Cost

If the values in the verification tuple *u* are random with uniform distribution, nearly half of those values will be encoded using the chains from the stack. And the other half will be encoded using chains from the stream. Verification will require nearly half the number of hash function iterations. Then, the synced signature will have the same length as the initial signature, but nearly half the expected verification cost.

3.3 Security

Since the values in the *u* tuple are encoded from top to bottom (figure 3), the *u* values known from the synced signature are only useful to compute smaller *u* values.. So, the synced signature is secure as long as the original OTS is secure (from lemma 1, section 2.1).

In fact, the range of known values is smaller in the synced signature than in the original OTS.

The positions (letters) within the *u* tuple that the chains in the stream encode are determined by the used chains in the stack.

Since the length of the signature is constant, a fixed number of chains must be used. To modify those positions, an attacker would need to discard a known value and publish an additional, unused value from a different chain. Either from the stack or from the stream. In both cases, a forgery would require a value that is not known, and cannot be obtained without breaking the hash function.

In other words:

> If the signature consists of a fixed number of hashes, an attacker cannot replace values from chains at given positions (in the stack or the stream) to a different set of positions without breaking the hash function **(lemma 2)**.

Synced State

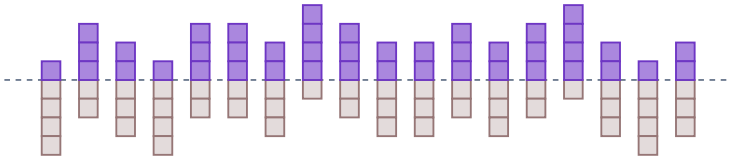
The signatures is stateful, but the the signer does not need to store the state. Signer and verifiers synchronize the state of the signature using the blockchain:

By observing the blockchain, the signer and the verifiers can recover the number of signatures, the hash values that were spent and cannot be reused, and the hash values that remain unspent and can be used to authenticate a new message.

4. Generalization to Many Signatures

We want to extended the scheme from two signatures to many. We will authenticate each signature using the unspent values from all previous chains.

4.1 Stack



(Figure 5)

We'll keep a stack, containing all chains with unspent values from previous signatures (figure 5).

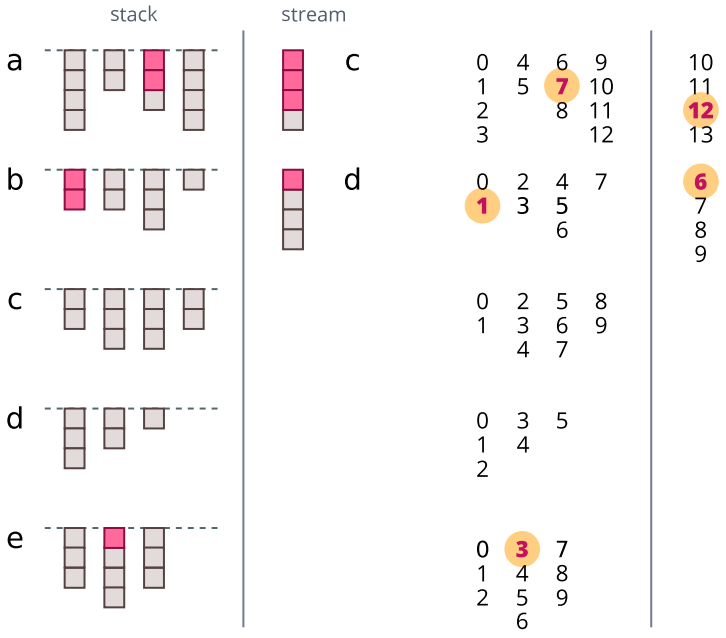
Each chain has an spent part (in purple, above the line), that is already known and cannot be reused. And an unspent, unknown part, that ranges from the private key to the preimage of the last known hash. The chains are sorted in the stack in the same order they had in the stream.

After each signature, we will discard the chains that are completely spent.

4.2 Signing

As the number of signatures grows, we'll have more chains in the stack. To make the signature more efficient, we want to use all of them to encode the *u* tuple. So, rather than encoding the values at a position (letter) from *u* using a single chain, we will split those values among multiple chains in the stack.

We will start by grouping the chains in the stack as evenly as possible. And we'll assign each group to a position in the *u* tuple, indicated by a letter (figure 6, left). Each row in the figure contains a group of chains, and its letter.



(Figure 6)

Then, we'll assign the possible values in the *u* tuple to the hashes in each group (figure 6, right). From top to bottom in each chain, and from the first chain in a group to the last. The same as before, we'll use the stream if needed to encode the remaining values.

In the example, a tuple *u = (7, 1, 12, 6, 3)* is mapped to the synced signature as follows:

u = (7, 1, 12, 6, 3) ⟶
u_{stack}, u_{stream}, pos = (2, 2, 1), (2, 0), (2, 4, 16)

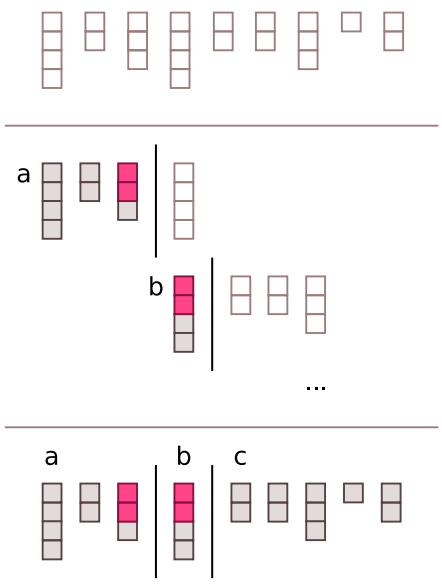
4.3 Security

The security follows the same principles as before:

From the values in the synced signature, and attacker could only compute smaller *u* tuple values. So, the signature is secure as long as the original OTS is secure (from lemma 1, section 2.1).

Like before, the letters that correspond to the chains in the stream are determined by the used chains in the stack. Since the number of hashes in the signature is fixed, an attacker cannot change the positions without breaking the hash function (from lemma 2, section 3.3).

5. Adaptive Encoding



(Figure 7)

To encode the verification tuple *u*, we have split the stack into groups, and used a single chain from each group.

As a result, there are combinations of chains that don't encode any message. For example, a signature taking values from two chains in the first row (at position *a*) has no meaning.

We want to use all possible combinations of chains, to maximize the number of distinct tuples the stack can encode. To achieve that, we'll use an adaptive method, where the values assigned to each chain depend on the chain that was used before.

The method (figure 7) works as follow:

Suppose we want to encode a tuple with length L_u, and we have a stack with a number of chains L_{stack} .

To encode the first position in u, we'll use a number of chains L_a from the stack, given by:

$L_a = ceiling\ (L_{stack} / L_u)$

We'll assign the hashes in those L_a chains to encode the value from the u in the same way as before, using the stream if necessary.

Now, suppose we use the chain at position n in the stack, counting from zero. The following chains ($n+1, n+2, ...$) are not needed for this encoding. So, we will reuse them to encode the next value from u.

We will recompute the number of remaining chains from the stack, and the remaining values to encode from the tuple u:

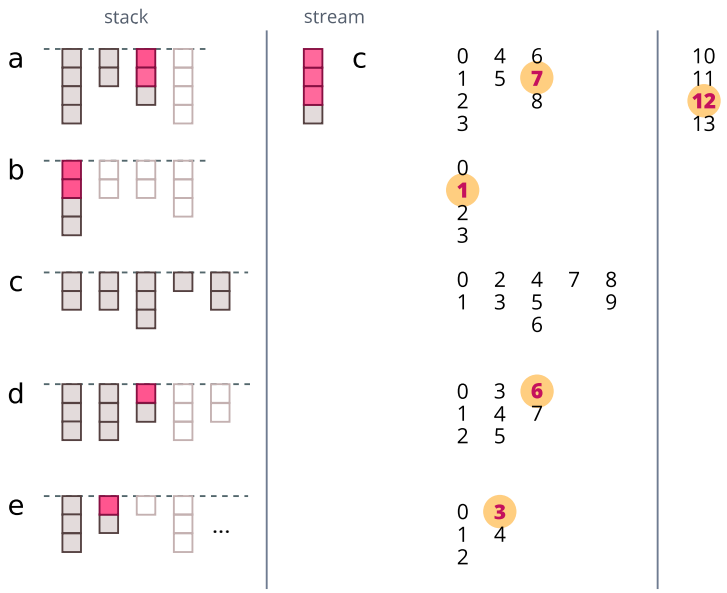
$L'_{stack} = L_{stack} - (n + 1)$
$L'_u = L_u - 1$

Now, we'll recompute L_a, using the same equation as before. We'll take the next L_a chains, starting from position n to encode the next value from u.

We will repeat the procedure until all values from u are encoded, as illustrated in figure 8.

The mapping in our example has become:

u = (7, 1, 12, 6, 3) ⟶
u_{stack}, u_{stream}, pos = (2, 2, 1, 1), (2), (2, 3, 11, 13)



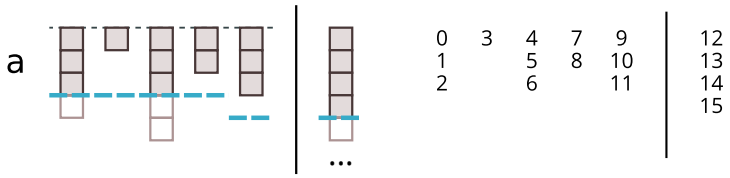
(Figure 8)

5.1 Security

Now, the position of each used chain from the stack modifies the value the next chain encodes.

Since the length of the signature is constant, to modify the position an attacker would need to publish a value from a different chain, either from the stack or the stream. Like before, the attacker needs a value that is unknown, and cannot be obtained without breaking the hash function (from lemma 2, section 3.3).

6. Balanced Chains



(Figure 9)

When we assign the possible values from the *u* tuple to the hashes in the stack, it is possible that all values end up encoded in the first few chains. To minimize the cost of verification, we want them to be as evenly distributed as possible among the chains.

For that purpose, we will limit the amount assigned to each chain. Suppose we have *g* values to encode (*0...15*). And L_a*+1* chains to encode them, L_a from the stack and an extra one from the stream if needed. We will limit the values a single chain can encode to:

limit = ceiling (g / (L_a *+ 1))*

For example, if we have 16 values to encode (*0...15*), 5 chains from the stack, and an extra chain from the stream, we will assign at most 3.

Since the number of unspent hashes in each chain is random, the number may also be smaller than ideal. So, rather than applying the same limit to all chains, we will recompute the limit for each. In our example, after assigning the first 3 values (*0, 1, 2*), we'll update the number of remaining values and remaining chains, and we'll use the same equation to recompute the limit.

After assigning 3 values, we have 13 values and 4+1 chains remaining. The next limit becomes:

limit = ceiling (13 / (4 + 1)) = 3

We will repeat the procedure, until all values are assigned to the chains, as in table 1.

remaining	L_a+1	limit	unspent	assigned
16	5+1	3	4	3
14	4+1	3	1	1
13	3+1	3	5	3
10	2+1	3	2	2
8	1+1	4	3	3
5	0+1	4	16	4

(Table 1)

7. Calculations

We'll use a pure python implementation to estimate the cost of the synced many-times signature, and see how it compares against the OTS it is made of.

We'll keep track of the number of unspent values in each chain in the stack, that we'll use to authenticate the signatures. Our main function will take as input a verification tuple that corresponds to a message, and the unspent stack list. It will return the synced verification tuples (from the stack and from the stream), and an updated stack list.

The cost of verification (measured in hash iterations) is the sum of the verification tuples.

7.1 Code

The *sync1up* function syncs a single value from the verification tuple to the stack.

It takes as input that value *(up)*, the number of unspent values in the stack, and the constant *zeta* that indicates the maximum value *up* can take.

It returns the synced value, the remaining unspent values in the stack, and the number of unused chains.

```
def sync1up (up, stack, zeta):
        m = -1
        for n, s in enumerate (stack):
                m += s
                if m >= up:
                        # stack
                        r = m - up
                        u = s - r
                        stackUsed = stack[0:n] + [r]
                        L_unused = len (stack) - (n + 1)
                        return u, stackUsed, L_unused
        # stream
        r = m + zeta + 1 - up
        u = zeta - r
        stackUsed = stack + [r]
        L_unused = 0
        return u, stackUsed, L_unused
```

The *balance* function takes as input the list *stack* with the number of unspent values in a set of chains from the stack, and the *zeta* constant.

It returns a balanced stack, and the remaining values.

```
def balance (stack, zeta):
        balanced = []
        rest  = []
        ups = zeta + 1
        L = len (stack) + 1
        for n, s in enumerate (stack):
                t = min (s, ups // (L - n))
                balanced += [t]
                rest  += [s - t]
                ups -= t
        return balanced, rest
```

The *balanced_sync1up* combines the *balance* and *sync1up* functions. It balances some chains from the stack, syncs a value from the verification tuple, and adds the remaining values from balancing back.

```
def balanced_sync1up (up, stack, zeta):
        # balance
        stackB, rest = balance (stack, zeta)
        # sync
        u, stackUsed, L_unused = sync1up (up, stackB, zeta)
        # add rest
        L = len (stack) - L_unused
        for n in range (L): stackUsed[n] += rest[n]
        return u, stackUsed, L_unused
```

Th function *sync_signature* puts all other functions together. It syncs a signature, defined by its verification tuple, to the stack.

It takes as input the verification tuple *ups* with each part in the range *0...zeta*, the list of unspent values in each chain of the stack, and the constant *zeta*.

It returns the synced version of the tuple *ups*, and the updated number of unspent values in the stack.

```
def sync_signature (ups, stack, zeta):
        ups2 = []
        stack2 = []
        pos = 0
        for n, up in enumerate (ups):
                # take chains from the stack
                num, den = len (stack) - pos, len (ups) - n
                L = (num + den - 1) // den
                stackA = stack[pos:pos+L]
                # sync
                u, stackA, L_unused = balanced_sync1up (up, stackA, zeta)
                ups2 += [u]
                stack2 += stackA
                # update position
                pos += L - L_unused
        # append remaining
        stack2 +=  stack[pos:]
        # discard empty chains
        stack2 = [n for n in stack2 if n > 0]
        return ups2, stack2
```

7.2 Simulation

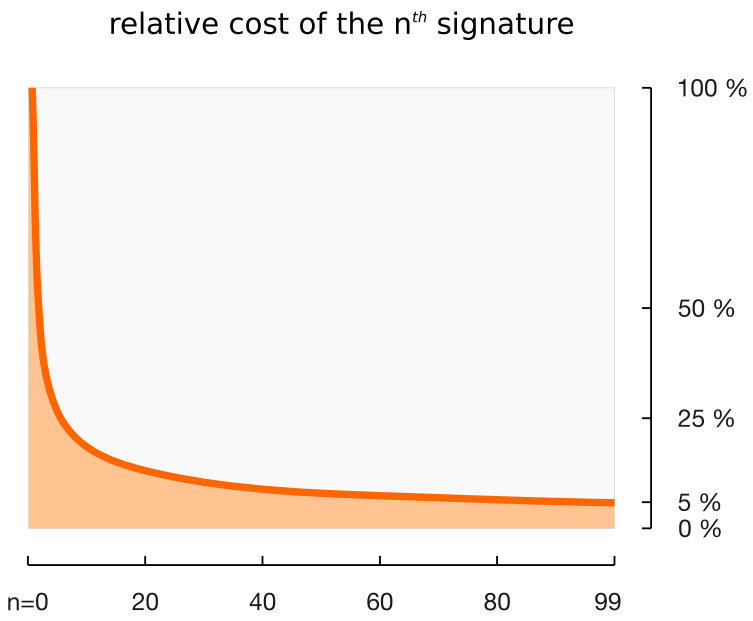
A signer will use the scheme to sign a sequence of many messages $m_0, m_1, m_2, \ldots$. We want to estimate the cost of each signature in the sequence, compared to the OTS the scheme is built from. We will measure the cost as the number of hash iterations needed to move up in the chains. The initial, zeroth signature will have the same cost as the OTS. And each consecutive signature is expected to cost less than the signatures before.

Rather than signing real messages, we will run a simulation using *u* tuples generated at random. We'll use tuples with length 18 and each part in the range $0...2^{12}-1$. Those values were chosen to match a Winternitz OTS with parameters *bits=192* and *w=12*. The simulation, like the scheme, starts with an empty stack that grows with each signature.

Using the code to estimate the cost of 100 signatures, we have:

```
from random import randrange
length = 18
zeta = 2\*\*12 - 1
signatures = 100
cost_ots = zeta \* 0.5 \* length
costs = []
stack = []
for n in range (signatures):
    u = [randrange (zeta+1) for n in range (length)]
    us, stack = sync_signature (u, stack, zeta)
    cost_sync = sum (us)
    c = cost_sync / cost_ots
    costs += [c]
print costs
```

We will run a large number of simulations and average the cost from all samples. to estimate to expected cost of each of the 100 signatures, from the 0*th* to the 99*th*. Figure 10 shows the results.



(Figure 10)

As the plot shows, the cost reduction can be quite drastic: the 99*th* signature costs only 5.8% compared to the initial OTS.

Table 2 shows numerical values for some of the signatures.

n	relative cost
0	100 %
1	58 %
10	18.1 %
20	12.9 %
40	9.1 %
60	7.5 %
80	6.4 %
99	5.8 %

(Table 2)

8. Futher Improvements

8.1 Limited Stack

We could impose a limit to the size of the stack, to prevent it from growing. To achieve that, we could trim the stack after each signature and keep a maximum number of chains, the ones with most unspent values.

8.2 Compressed Blockchain

As each chain from the stack is used multiple times to sign many messages, many intermediate hashes are published. Those values can be easily computed by iterating the hash function over the known value at the bottom of the chain. Therefore, they can be omitted from the blockchain to reclaim space.

Previous h_n values can be omitted as well.

A compressed blockchain would require storing all messages, but only the hashes from the latest signatures. Older signatures can be scrapped from the blockchain, since they can be easily recreated from the latter ones.

Exhausted Stream

Usually, private keys are derived from a single private value (a private seed), using a cipher or a hash function in counter mode.

If the the stack is limited, once all chains from a same stream are discarded, its private seed can be published to the blockchain. Once inserted in a block, it is possible to further shrink the size of all signatures to the size of a single hash.

8.3 Improvements to Specific OTSs

In this paper we have introduced a general method to transform an OTS into a many-times scheme. Usually, we can apply other improvements that are specific to a particular OTS. A Winternitz OTS is a simple example.

The *u* tuple in a Winternitz OTS signature is composed of two parts:

u = (a, b, c, ...) || (k, l, m, ...)

The left part represents the message. The right part is a checksum needed to prevent forgeries. The size of the checksum depends on the number of values an attacker could compute from the left part, from the signature to the public key. Since the range is smaller in the synced signature, the checksum can be made smaller as well.

9. Conclusions

Previously known many-times schemes grow in size and cost as the number of needed signatures grows. This can make the signatures impractical in real-world applications.

We have shown that is possible to transform an OTS into a many-times signature that is more efficient than the OTS it is made of, rather than less.

The proposed scheme permits an unlimited number of signatures. The size of the signatures is constant, and the same as in the OTS. The verification cost starts the same as in the OTS and decreases with each new signature, becoming more and more efficient as the number of signatures grows.

References

1. Leslie Lamport - Constructing Digital Signatures from a One Way Function (1979).
2. Ralph C. Merkle - Secrecy, Authentication, and Public Key Systems (1979).
3. Santi J. Maccallini - Integer Compositions Signatures (2016).
4. Ralph C. Merkle - A Certified Digital Signature (1989).
5. Satoshi Nakamoto - Bitcoin: A Peer-to-Peer Electronic Cash System (2008).