Integer Compositions Signatures

Santi J. Vives Maccallini
jotasapiens.com/research/

Abstract: We introduce integer compositions signatures (ic): a post-quantum, hash-based family of one-time signatures. The proposed scheme explores a connection between hash-based signatures and combinatorics: the authentication path taken from the signature to the public key is determined by a restricted composition of an 9. The family shows improvements over previous schemes like Winternitz: reduced cost, verification in constant time, and the possibility to tweak the signature for either faster signing or faster verification.

Keywords: one-time signatures, ots, hash, authentication, post-quantum cryptography, composition, combinatorics.

1. Introduction

In this paper we will introduce **ic**, a family of one-time [1], hash-based digital signatures.

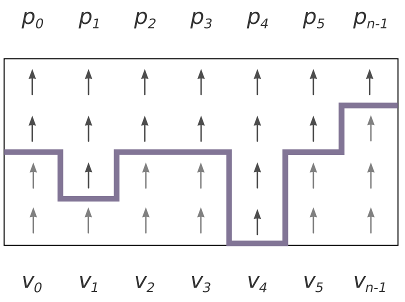
The family shows improvements over previous hash-based schemes like wots (Winternitz one-time signatures). Some of the advantages are:

- A more efficient size/cost trade-off, allowing for signatures of the same size with less computational cost (fewer hash functions evaluations) or a similar cost and a smaller size.
- Verification in constant time and signing in nearly constant time.
- Resistance against forgery without the need for checksums.
- Tweakable parameters, that make the signature tunable to a large range of uses:
    - Unlike wots, whose *w* parameter only leads to signatures with different (but limited) signature lengths, the size of the signatures can by adjusted to an arbitrary number of output hashes *L (length)*.
    - In addition to *compression*, the ic family allows for *expansion* to reduce the cost of signing and verifying.
    - The signature can be tweaked for fast verification, fast signing, and values in between.

1.1 Concepts

Winternitz One-Time Signatures (wots)

In a Winternitz (wots) scheme [2][3], the signer picks *n* random numbers to create the private key *v* (at the bottom of the graph), each number with size *bits*.



Then, a keyed hash function is iterated over each number at the bottom to compute the public key *p* at the top. The one-wayness of the hash function ensures the values at a lower level cannot be computed from higher one.

In order to sign, the hash of a message is encoded as a list f_m of *w*-bit numbers. The parameter *w* determines the level of compression of the signature. The hash function is iterated over the first numbers in the private key *v*, a number ot times determined by f_m.

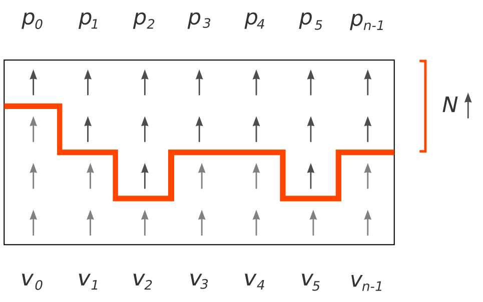
Once the signature is published, all values at higher levels (those between f_m and the public key) become public. To avoid an attacker from forging a signature, a checksum of the signature is needed. The checksum is computed in a similar way as the main part of the signature.

To verify, the iterations remaining to reach the higher level are applied to the main part of the signature and to the checksum. The signature is valid if the result matches the public key *p*.

Integer Compositions (ic)

An ic signature represents each message as an integer composition of an integer N. That is, one of the many ways an integer N can be written as a sum of positive parts, taking into account the ordering of the parts.
For example, the tuples *(2, 2, 2, 2)*, *(4, 2, 1, 1)*, *(1, 2, 2, 3)* and *(3, 2, 1, 2)* are different compositions of the integer *8*.

Similarly to wots, each parts in the composition is used to determine the number of iterations of the hash function. But a difference arises: since the composition requires that their parts add up to N (as seen in the graph), a higher number of iterations in any of the parts results in a lower number of iterations in at least another.



An attacker can no longer forge a signature from the values of a known message without breaking the one-wayness of the hash function. This eliminates the need for a checksum, leading to a smaller, faster signature.

The ic family of signatures uses compositions that are length-restricted (with a fixed number of parts), and alphabet-restricted, with each part taking values from an alphabet *0, 1, ..., zeta*.

The *(N, zeta)* pair of constants allows to tweak how the cost is distributed between the signer and the verifier. As *N* becomes smaller (and *zeta* higher) the cost of verifying decreases. As *zeta* becomes smaller (and *N* higher) the cost of signing and keys creation decreases.

In the different sections of this paper, we will:

- describe a method to transform the hash of a message into a restricted composition, equivalent to picking a composition uniformly at random from the set of all possible compositions **(2.3, 2.4, 2.5)**..
- describe the signature scheme based on restricted integer compositions. **(2.2, 2.3, 2.4)**.
- describe methods for finding optimal *(N, zeta)* constants, that minimize the number of hash evaluations for different uses **(2.1)**.
- compare different types of ic signatures, showing that the family can be tweaked to outperform wots verification or signing **(4.)**.

2. Description of the algorithm

2.1 Parameters and constants

The ic signature takes 3 main parameters: *bits, length* and *type*.

An ic signature with parameters *(bits, length)* is a one-time signature of size=*length*, capable of signing at least 2^{bits} distinct messages.

Let the auxiliary function *R (N, L)* be the amount of all possible compositions of the integer *N* with length *L*. **(2.5)**

Let the auxiliary function *R (N, L, z)* be the number of all possible compositions of the integer *N* with length *L* and each part taking values from the range *0...z*. **(2.5)**

Type 'v'

(Optimal for fast verification)

Given the parameters *bits* and *length*:

1. Find the smallest positive integer *N* that satisfies:
   $R(N, length) \geq 2^{bits}$
2. Find the smallest positive integer *zeta* that satisfies:
   $R(N, length, zeta) \geq 2^{bits}$
3. Return the constants *(N, zeta)* as the result.

Type 'a'

(Approximately optimal for verifying, while being faster at keys creation and signing)

Given the parameters *bits* and *length*, and a *tolerance* value:

1. Find the smallest positive integer *N'* that satisfies:
   $R(N', length) \geq 2^{bits}$
2. Compute *N*:
   $N = N' \cdot (1 + tolerance / 100)$
3. Find the smallest positive integer *zeta* that satisfies:
   $R(N, length, zeta) \geq 2^{bits}$
4. Return the constants *(N, zeta)* as the result.

Type 's'

(Optimized for faster keys creation and signing)

Given the parameters *bits* and *length*:

1. Find the smallest positive integer *zeta* that satisfies:
   $W(zeta, N_{zeta}, r_{zeta}) < W(zeta+1, N_{zeta+1}, r_{zeta+1})$
2. Define N:
   $N = N_{zeta}$
3. Return the constants *(N, zeta)* as the result.

Where:

- The cost function *W* is given by:
  $W(z, n, r) := w_{key} + w_{sig} + w_{comp}$
  And,
  $w_{key} = z \cdot length$
  $w_{sig} = z \cdot length - n$
  $w_{comp} = R(n, length) / r$
- N_z is the smallest positive integer *n* that satisfies the following condition, if such integer exists for a given *z*:
  $R(n, length, z) \geq 2^{bits}$
- And r_z is given by:
  $r_z := R(N_z, length, z)$

Type '1/s'

(Optimal for fast keys creation and signing. Inverse mode)

The constants *(N, zeta)* are computed in the same way as in type 's'. But the cost function *W* is defined as follows:

- $W(z, n, r) := w_{key} + w_{sig} + w_{comp}$
  Where,
  $w_{key} = z \cdot length$
  $w_{sig} = n$
  $w_{comp} = R(n, length) / r$

M and mbits constants

Given the parameter *length* and the constant *N*:

Compute the constant *M*:

$$M = \prod_{n=1}^{length} (N + n)$$

Compute the constant *mbits*, given by the amount of bits needed to encode the integer *M − 1*:

$mbits = len\ (binary\ (M - 1))$

hashA, hashUp

Choose a keyed hash function *hashA* with output size *mbits*, and a keyed hash function *hashUp* with output size *bits*.

2.2 Keys creation

1. Generate the private key by picking a list of random numbers, each number with size *bits*:
   $priv := priv_0, priv_1, priv_2, \ldots, priv_{length-1}$
   $priv_n = urandom\ (bits)$
2. Generate a salt by picking a random number with size ≥ *bits*:
   $sal = urandom\ (bits)$
3. Iterate the hash function over each value in the private key. The number of iterations is given by *zeta*. Each hash function is keyed with a unique number, created from the salt, a column index, and a row index:
   $f := f_0, f_1, \ldots, f_{length-1}$
   - For $n = 0, 1, \ldots, length - 1$:
     - $f_n = priv_n$
     - For $i = 0, 1, \ldots, zeta$:
       - $s = sal\ ||\ n\ ||\ i$
       - $f_n = hashUp_s\ (f_n)$
4. Compress f_n:
   $pubcheck = hashUp_{sal}\ (f_0\ ||\ f_1\ ||\ \ldots\ ||\ f_{length-1})$
5. Publish *pubcheck* as the public key. Keep *priv* and *sal* secret.

2.3 Signing

Given a *message*, a private key, and a salt:

1. Compute the hash value *h* of the message: $h = hashA_{sal}\ (message)$
2. Given a counter $n = 0, 1, \ldots$
   - Compute:
     $m = hashA_{sal}\ (n\ ||\ h)$
   - Compute the *m*-th restricted composition of *N*, using the auxiliary function:
     $c = composition\ (N, length, index=m)$
   - Stop the counter when a pair *(m, c)* is found that satisfies:
     $m < M$
     $max\ (c) \leq zeta$

3. Compute the tuples *upsVer* and *upsSig*:
   $upsVer = c_0, c_1, \ldots, c_{length-1}$
   $upsSig = zeta-c_0, zeta-c_1, \ldots, zeta-c_{length-1}$
4. When using the inverse mode, swap the tuples:
   - If *type* == '*1/s*':
     - *upsVer, upsSig = upsSig, upsVer*
5. Iterate the hash function over each value in the private key. The number of iterations is given by *upsSig*. Each hash function is keyed with a unique number, created from the salt, a column index, and a row index:
   $f := f_0, f_1, \ldots, f_{length-1}$
   - For $n = 0, 1, \ldots, length - 1$:
     - $f_n = priv_n$
     - For $i = 0, 1, \ldots, upsSig_n - 1$:
       - $s = sal \,||\, n \,||\, i$
       - $f_n = hashUp_s\,(f_n)$
6. Publish *(f, n, sal)* as the signature.

2.4 Verification

Given a *message*, a signature *(f, n, sal)* and a hashed public key *pubcheck*:

1. Compute the hash value h of the message:
   $h = hashA_{sal}\,(message)$
2. Compute the hash value m of the message h and the counter n:
   $m = hashA_{sal}\,(n \,||\, h)$
3. Check that $m < M$.
4. Compute the m-th restricted composition of N, using the auxiliary function:
   c = composition (N, length, index=m)
5. Check that no part in the composition is bigger than *zeta*:
   $max\,(c) \leq zeta$
6. Compute the tuples *upsVer* and *upsSig*:
   $upsVer = c_0, c_1, \ldots, c_{length-1}$
   $upsSig = zeta-c_0, zeta-c_1, \ldots, zeta-c_{length-1}$
7. When using the inverse mode, swap the tuples:
   - If *type* == '*1/s*':
     - *upsVer, upsSig = upsSig, upsVer*
8. Iterate the hash function over each value f_n in the signature. The number of iterations is given by *upsVer*. Each hash function is keyed with a unique number, created from the salt, a column index, and a row index:
   - For $n = 0, 1, \ldots, length - 1$:
     - For $i = 0, 1, \ldots, upsVer_n - 1$:
       - $s = sal \,||\, n \,||\, upsSig_n + i$
       - $f_n = hashUp_s\,(f_n)$
9. Compress f_n:
   $t = hashUp_{sal}\,(f_0 \,||\, f_1 \,||\ldots\, ||\, f_{length-1})$
10. Verify that $t == pubcheck$.
11. The signature is valid if all tests (3, 5 and 10) evaluate to *True*, invalid otherwise.

2.5 Auxiliary functions

composition

Transforms and integer m in the range $0 \ldots M-1$ into a length-restricted composition of N. The constant M is:
$M = N \cdot (length - 1)!$

Where *!* is the factorial function.

Given the constants N, *length* and an index m:

1. Represent the integer m as a mixed-radix number with bases b:
   $b = N+length-1, N+length-2, \ldots, N-1$
   d = mixradix (m, bases=b)
2. Create the tuple p:
   $p = (p_0)$
   $p_0 = N + length - 1$
3. For each d_n in d:
   - Define r:
     $r = d + 1$
   - Given a counter $n = 0, 1, \ldots$
     - If $r - p_n \leq 0$, stop the counter. Let $\tilde{n}$ be the last value of the counter.
     - Else, subtract p_n from r:
       $r \mathrel{-}= p_n$
   - Append $(p_{\tilde{n}} - r)$ to p:
     $p = p \,||\, (p_{\tilde{n}} - r)$
   - Assign $(r - 1)$ at index $\tilde{n}$ of p:
     $p_{\tilde{n}} = r - 1$
4. Return the tuple p as the result.

mixradix

Given a *number* and a list of bases *b*:

```
1. Set the variable $n$:
   $n = number$
2. Create the empty tuple $r$:
   $r = (\,)$
3. For each $b_{last}, .., b_1, b_0$:
     • Compute
       $e = n \bmod b_n$
       $n = floor(n / b_n)$
     • Append $e$ to the tuple $r$:
       $r = r \,||\, e$
4. Return $r$ as the result.
```

R (N, L)

The number of restricted integer compositions of *N* with exactly *L* parts, and each part taking values in the range *0...N*:

$$R(N, L) = \binom{N+L-1}{L-1}$$

Where $\binom{b}{c}$ is the binomial coefficient.

Given the parameter *L* and *N*:

```
1. Set the variable $r$:
   $r = 1$
2. For $n = 1, 2, .., L$:
     • $r = r \cdot (N + n)$
3. Compute:
   $r = r / (L - 1)!$
4. Return $r$ as the result.
```

Where *!* is the factorial function.

R (N, L, z)

The number of restricted integer compositions of *N* with exactly *L* parts, and each part taking values in the range *0...z*. Given by [4]:

$$R(N, L, z) = \sum_{n=0}^{L} (-1)^n \cdot \binom{L}{n} \cdot \binom{N+L-1-n\cdot(z+1)}{L-1}$$

Where $\binom{b}{c}$ is a non standard binomial coefficient, defined as 0 (zero) for any $b < 0$ and $b < c$.

3. Table of N, zeta constants

The table shows computed *N, zeta* values for parameter *bits=256*, with different *lengths* and *types*.

length	type 'v'	type 'a'	type 's'
20	N=90189 zeta=41972	N=91541 zeta=18786	N=120841 zeta=12105
24	N=21126 zeta=7730	N=21442 zeta=3707	N=28017 zeta=2369
28	N=7796 zeta=2316	N=7912 zeta=1183	N=9903 zeta=755
32	N=3786 zeta=962	N=3842 zeta=507	N=4613 zeta=326
40	N=1437 zeta=318	N=1458 zeta=157	N=1665 zeta=103
48	N=778 zeta=112	N=789 zeta=72	N=868 zeta=49
56	N=510 zeta=67	N=517 zeta=42	N=557 zeta=29
64	N=375 zeta=45	N=380 zeta=28	N=409 zeta=19

4. Evaluation

To evaluate the scheme, we will take into account the cost of signing and verifying, defined as the number of hash evaluations. The cost of verification is given by W_{ver}, and the cost of signing corresponds to the added costs W_{comp} and W_{sig} (the cost of finding a valid composition, and the cost of computing the signature from the private key):

$W_{ver} = N$
$W_{sig} = N \cdot (zeta - 1)$
$W_{comp} = R(N, length) / R(N, length, zeta)$

The table compares the cost of 256-bit ic signatures with length=28 and different types.

	size (bits)	signing	verification
ic 256 - type 'v'	7424 bits	57.1 ms_h	**7.8** ms_h
ic 256 - type 'a'	7424 bits	25.2 ms_h	**7.9** ms_h
ic 256 - type 's'	7424 bits	10.5 ms_h	9.9 ms_h
ic 256 - type '1/s'	7424 bits	**9.4** ms_h	12.3 ms_h
wots+ 256	7424 bits	14.3 ms_h	14.3 ms_h

For reference, a ms_h equals 1 ms, assuming a computer performing 1 million hash iterations per second.
The length in bits includes the size of the salt used to randomize the hash functions.

5. Source code

A pure python implementation of the signature scheme is provided. The work can be found at:

1. http://jotasapiens.com/research

Current version: Hash-based signatures (ic, icvar, uwots) version 4 (2016 Oct 24).

sha256:
754a600b4cef06d5c773440eb9ed1bc0e3ce2088273997
03ededd833a141a0fc

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. Andreas Hülsing - On the Security of the Winternitz One-Time Signature Scheme (2013).
4. Morton Abramson - Restricted Combinations and Compositions (1976).