Integer Composition Signatures

(Draft)

Santi J. Vives Macallini
@jotasapiens
http://jotasapiens.com

Abstract: We introduce integer composition signatures (ic): a hash-based family of one-time signatures. The family shows improvements over previous schemes like Winternitz: less costly/shorter signatures, verification in constant time, and tweakable parameters allowing optimization for either signing/verifying.

Keywords: signatures, hash, postquantum, cryptography.

1. Introduction

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

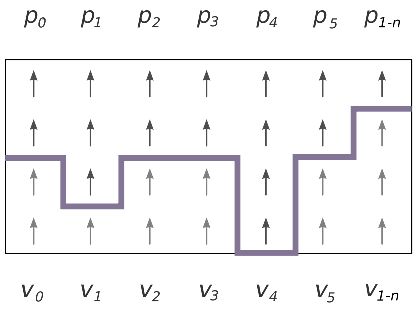
The family shows improvements over previous hash-based schemes like wots (Winternitz one-time signatures). Some of the advantages are:

- 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 with a signature of 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 of various (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 tuned for fast verification, fast signing, and values in between.

1.1 Concepts

Winternitz One-Time Signatures (wots)

In a wots scheme, the signer picks *n* numbers uniformly at random to create the private key *v* (at the bottom of the graph).



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

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

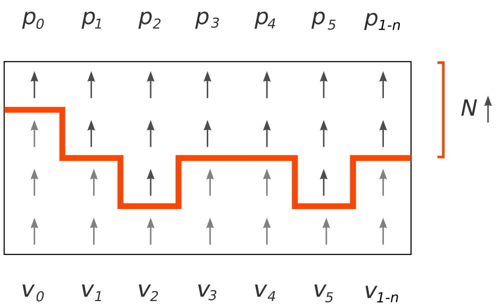
Once the signature is published, all values at higher levels (those between f_m and the public key) become known. To avoid an attacker from forging a signature, a checksum of the signature is needed. The checksum is computed in a similar way as the main part of the signature.

To verify, the iterations remaining to reach the higher level are applied to the main part and the checksum. The result is compared against the public key *p*.

Integer Composition Signatures (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 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 distinct compositions of the integer 8.

Similarly to wots, the parts in the composition are used to determine the number of iterations of the one-way functions. But a difference arises: since the composition requires that their parts add up to N (as seen in the graph), a higher level in any of the parts results in a lower level in at least another.


p_0 p_1 p_2 p_3 p_4 p_5 p_{1-n}
N
v_0 v_1 v_2 v_3 v_4 v_5 v_{1-n}

An attacker can no longer forge a signature from the values of a known message without breaking the one-way 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 parts taking values from an alphabet=*0, 1, 2, ..., zeta*.

In the different sections of this paper, we will:

- describe a method to transform the hash of a message into a restricted composition, which is 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 constants, that minimize the number of hash evaluations for diferrent uses **(2.1)**.
- compare various 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 amount of all possible compositions of the integer *N* with length *L* and a restricted alphabet=*0, 1, 2, ..., z*. **(2.5)**

Type 'v' constants

(Optimal for fast verification)

Given the parameters *bits* and *length*:

1. Using the auxiliary function *R*, find the smallest integer *N* that satisfies:
   $R (N, length) >= 2^{bits}$
2. Find the smallest integer *zeta* that satisfies:
   $R (N, length, zeta) >= 2^{bits}$
3. Return the constants *(N, zeta)* as the result.

Type 'a' constants

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

Type 'zmin' constants

(Minimal keys creation time, at the expense of greater signing time)

Given the parameters *bits* and *length*:

1. Define the function *n (z)* as the smallest integer *n* satisfying:
   $R (n, length, z) >= 2^{bits}$
2. Find the smallest integer *zeta* that satisfies:
   $R (n(zeta), length, zeta) >= 2^{bits}$
3. Define *N*:
   $N = n(zeta)$
4. Return the constants *(N, zeta)* as the result.

Type 's' constants

(Optimized for fast keys creation and signing)

Given the parameters *bits* and *length*:

1. Define the cost function *W (nz, nn, nr)*:
   $w_{keys} = nz * length$
   $w_{sign} = nz * length - nn$
   $w_{comp} = R (nn, length) / nr$
   $W (nz, nn, nr) = w_{keys} + w_{sign} + w_{cost}$
2. Define the function *n (z)* as the smallest integer *n* satisfying:
   $R (n, length, z) >= 2^{bits}$
3. Define the function *r (z)* as:
   $r (z) = R (n(z), length, z)$
4. Find the smallest integer *zeta* that satisfies:
   $W (zeta, n(zeta), r(zeta)) < W (zeta+1, n(zeta+1), r(zeta+1))$
5. Define *N*:
   $N = n(zeta)$
6. Return the constants *(N, zeta)* as the result.

Type '1/s' constants

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

Given the parameters *bits* and *length*:

1. Define the cost function *W (nz, nn, nr)*:
   $w_{keys} = nz * length$
   $w_{1/sign} = nn$
   $w_{comp} = R (nn, length) / nr$
   $W (nz, nn, nr) = w_{keys} + w_{1/sign} + w_{comp}$
2. Define the function *n (z)* as the smallest integer *n* satisfying:
   $R (n, length, z) >= 2^{bits}$
3. Define the function *r (z)* as:
   $r (z) = R (n(z), length, z)$
4. Find the smallest integer *zeta* that satisfies:
   $W (zeta, n(zeta), r(zeta)) < W (zeta+1, n(zeta+1), r(zeta+1))$
5. Define *N*:
   $N = n(zeta)$
6. Return the constants *(N, zeta)* as the result.

M and mbits constants

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

1. Set the variable *r*:
   $r = 1$
2. For $n = 1, 2, .., length$:
   - $r *= N + n$

3. Return $M = r$ as the result.

Given *M*:

1. Define the output size *mbits* of two one-way functions *hashA, hashB*:
   mbits = len (binary (M - 1))

2.2 Keys creation

Given a one-way function *hashUp* with output size *bits*:

1. Generate the private key by picking numbers (with *size=bits*) uniformly at random:
   $priv = priv_0, priv_1, priv_2,..., priv_{length\ -\ 1}$
   $priv_n = urandom\ (bits)$

2. For each $priv_n$, apply *zeta* iterations of the one-way function *hashUp*:
   $pub = pub_0, pub_1, pub_2,..., pub_{length\ -\ 1}$
   $pub_n = hashUp\ (priv_n, iterations=zeta)$

3. Publish the list *pub* as the public key.

2.3 Signing

1. Compute the hash value *h* of the message:
   h = hashA (message)

2. Given a counter *n = 0, 1, ...*

   - Compute:
     m = hashB (n || h)
   - Compute the *#m* restricted composition of *N*, using the auxiliary function *compR*:
     c = compR (N, length, i=m)
   - Stop the counter when a pair *(m, c)* is found that satisfies:
     m < M
     max (c) <= zeta

3. Compute the list *ups*, as needed for signing:
   $ups = ups_0, ups_1, ..., ups_{length-1}$
   Where each ups_n is given by:

   - If *mode == normal* (*type='v', 'a','s'* or 'zmin'):
     $ups_n = zeta - c_n$
   - Else, *mode == inverse* (*type='1/s'*):
     $ups_n = c_n$

4. Apply ups_n iterations of the one-way function *hashUp* to each $priv_n$ in *priv*:
   $f = f_0, f_1, ..., f_{length\ -\ 1}$
   $f_n = hashUp\ (priv_n, iterations=ups_n)$

5. Publish *(f, n)* as the signature.

2.4 Verification

Given a *message*, a signature *(f, n)* and a public key *pub*:

1. Compute the hash value *h* of the message:
   h = hashA (message)

2. Compute the hash value *m* of the message
   m = hashB (n || h)

3. Check that $m < M$.

4. Compute the *#m* restricted composition of *N*, using the auxiliary function *compR*:
   c = compR (N, length, i=m)

5. Check that no part in *c* is bigger than *zeta*:
   max (c) <= zeta

6. Compute the list *ups* (for verification):
   $ups = ups_0, ups_1, \ldots, ups_{length-1}$
   Where each ups_n is given by:

   - If mode == normal (*type*='*v*', '*a*', or '*s*'):
     $ups_n = c_n$
   - If mode == inverse (*type*='*1/s*'):
     $ups_n = zeta - c_n$

7. Apply ups_n iterations of the one-way function *hashUp* to each f_n:
   $t = t_0, t_1, \ldots, t_{length\ -\ 1}$
   $t_n = hashUp\ (f_n, iterations=ups_n)$

8. Check that *t == pub*.

9. The signature is valid if all test
   (steps 3, 5 and 8) evaluate to true, invalid otherwise.

2.5 Auxiliary functions

compR

(Transforms and integer *m* in the range *[0, M)* into a length-restricted composition of *N*)

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

1. Represent the integer m as a mix-radix number with bases b:
   $b = N+length-1, N+length-2, ..., N-1$
   $d = mixradix\ (m, bases=b)$
2. Create the tuple p:
   $p = (p_0)$
   $p_0 = N + length\ (d)$
3. For each d_n in d:
   - Define r:
     $r = d + 1$
   - Given a counter $n = 0, 1, ...$
     - If $r - p_n <= 0$, stop the counter. Let $\tilde{n}$ be the last value of the counter.
     - Else, subtract p_n from r:
       $r -= p_n$
   - 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)

$R\ (N, L)$ is the number of integer comps of N, with length L and alphabet $0, 1, 2, ..., N$

Given the parameter L and N:

1. Set the variable r:
   $r = 1$

2. For *n = 1, 2, .., L*:
    - *r *= N + n*

3. Compute:
   r /= factorial (L - 1)
4. Return *r* as the result.

R (N, L, z)

R (N, L, z) is the number of integer comps of *N*, with length *L* and alphabet *0, 1, 2, ..., z*

For a given *z*, the integer *R (N, L, z)* can be computed iteratively. To do so, we will define the special cases *L=1*, *N=0*, and a method to compute *R* for each *L = 2, 3, ...* from its previous value *L-1*:

- *R (N, **1**, z) = 1*
  for any value of *N, z*.

- *R (**0**, L, z) = 1*
  for any value of *L, z*.

- *R (N, L, z) = sum R (n, L-1, z)*, for *n = s0 ... s1*
  where:
  *p0 = max (0, N - z * (L- 1))*
  p1 = min (z, N)
  s0 = N - p1
  s1 = N - p0

- For a given *L, z* pair, *N* can take values in the range *[0, z*L+1).*

3. Table of N, zeta constants

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

length	type 'v'	type 'a'	type 's'	type '1/s'

20	N=90189
	zeta=41972
		N=91541
		zeta=18786
			N=120841
			zeta=12105
				N=109440
				zeta=12348

24	N=21126
	zeta=7730
		N=21442
		zeta=3707
			N=28017
			zeta=2369
				N=25613
				zeta=2419

28	N=7796
	zeta=2316
		N=7912
		zeta=1183
			N=9903
			zeta=755
				N=9316
				zeta=773

32	N=3786
	zeta=962
		N=3842
		zeta=507
			N=4613
			zeta=326
				N=4432
				zeta=334

40	N=1437
	zeta=318
		N=1458
		zeta=157
			N=1665
			zeta=103
				N=1621
				zeta=106

48	N=778
	zeta=112
		N=789
		zeta=72
			N=868
			zeta=49
				N=857
				zeta=50

56	N=510
	zeta=67
		N=517
		zeta=42
			N=557
			zeta=29
				N=549
				zeta=30

64	N=375
	zeta=45
		N=380
		zeta=28
			N=409
			zeta=19
				N=401
				zeta=20

(bits=256)

4. Evaluation

To evaluate the scheme we will take into account the cost of signing and verifying, given by 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_{sign} (the cost of finding a valid composition, and the cost of computing the signature from the private key):

$W_{ver} = N$
$W_{sign} = zeta * length - N$
$W_{ccomp} = R (N, length) / R (N, length, zeta)$

The following table compares the cost of keys creation for 256-bit ic signatures with length=28 and various types. Costs of wots+ are shown for comparison.

	length (bits)	**signing**	**verifying**
ic (type='v')	7424 bits	57.1 ms_h	**7.8** ms_h
ic (type='a')	7424 bits	25.2 ms_h	7.9 ms_h
ic (type='s')	7424 bits	**10.5** ms_h	9.9 ms_h
wots+	7424 bits	14.3 ms_h	14.3 ms_h

(bits=256)

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 a salt (or seed), used to randomize the hash functions.

5. Source code

A python implementation is provided to further illustrate the ic family of signatures. The code can be found at:

[1] http://jotasapiens.com/