# Unbalanced Winternitz Signatures

**(Draft)**

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

**Abstract:** We introduce 'uwots' (unbalanced Winternitz one-time signatures): an optimized, tweakable generalization of the Winternitz signature scheme.

*Keywords: signatures, hash, postquantum, cryptography.*

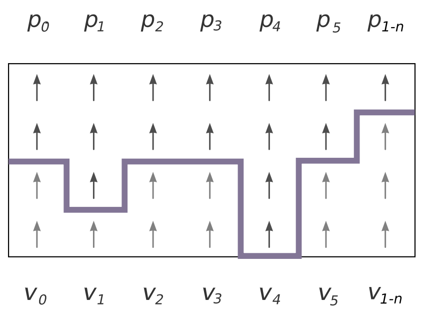
## 1. Introduction

In this paper we will introduce **uwots**, a family of one-time, hash-based, digital signatures. Uwots is an optimized, tweakable generalization of the Winternitz signature scheme.

### 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).



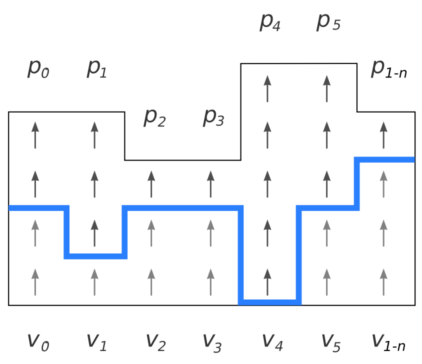
Then, a (keyed) one-way function is iterated over each of the numbers at the bottom to compute the public key *p* at the top. The one-wayness of the function ensures the values at a lower level cannot be computed from a higher one.

In order to sign, the hash of a message and a checksum are encoded as a list *f*, composed of *n* *w*-bit numbers. The parameter *w* determines the compression level of the signature. The one-way function is iterated over each part of the private key *v*, a number ot times determined by *f*.

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

**Unbalanced Winternitz One-Time Signatures (uwots)**

In the case of uwots, we encode the message and the checksum as mix-radix numbers instead. The different bases result in different (unbalanced) levels for different parts of the signature, as seen in the graph.



In the different sections of this paper, we will:

- describe the mix-radix (unbalanced) generalization of the Winternitz one-time signature scheme **(2.2, 2.3, 2.4)**.
- describe a method for finding optimal parameters, that minimize the number of hash evaluations **(2.1, 3.)**.
- show that the optimal parameters within the generalized form outperform wots, leading to signatures of the same size and reduced cost. **(4.)**

## 2. Description of the algorithm

### 2.1 Parameters and constants

Uwots takes 2 parameters: $L_p$ and *bits*.

An uwots signature with parameters *($L_p$, bits)* is a one-time signature of length=$L_p$ (measured in hash outputs), capable of signing at least $2^{bits}$ distinct messages.

**Computing necessary constant from the parameters**

Given the parameter $L_p$ and a *bits*:

1. List all the ways $s_n$ that an integer $L_p$ can be split in two parts: a main part *LAB* and a checksum *LCD*:
   $s = (1, L_p-1), (2, L_p-2), (3, L_p-3), \ldots, (L_p-2, 2), (L_p-1, 1)$
   $s_n = (LAB, LCD)$

2. For each $s_n$ = *(LAB, LCD)*:

   - Compute the bases *bA* and *bB*, given by:
     $bA = ceil\ (2^{bits} \wedge (1 / LAB))$
     $bB = bA - 1$
   - Find the smallest integer *LA* >= *0* that satisfies:
     bA ^ LA * bB ^ LB >= $2^{bits}$
     Where, for each *LA*, *LB* is given by:
     $LB = LAB - LA$
   - Compute *zA* and *zB*:
     $zA = bA - 1$
     $zB = bB - 1$
   - Compute the size of the checksum:
     $check = zA * LA + zB * LB$
   - Compute the bases *bC* y *bD*:
     $bC = ceil\ (check \wedge (1 / LCD))$
     $bD = bC - 1$
   - Find the smallest integer *LC* >= *0* that satisfies:
     bC ^ LC * bD ^ LD >= check
     Where, for each *LC*, *LD* is given by:
     $LD = LCD - LC$
   - Compute *zC* and *zD*:
     $zC = bC - 1$
     $zD = bD - 1$
   - Compute the cost *W* of $s_n$:
     $W = zA * LA + zB * LB + zC * LC + zD * LD$

3. Pick the $s_n$ value with the minimal cost *W*.

4. Return the constants *(LA, LB, LC, LD)*, *(bA, bB, bC, bD)* and *(zA, zB, zC, zD)* corresponding to the $s_n$ value we picked in step 3.

The constants $L_n$, $b_n$, $b_n$ are expanded into the lists *baseAB, baseCD, zetaAB, zetaCD, zeta* as following:

1. Define *list (c, length)* as a list with *length* copies of the element *c*. For example:
   *list (1, length=4) = 1, 1, 1, 1*

2. Define the lists *baseAB, baseCD, zetaAB, zetaCD* and *zeta* as following:
   *baseAB = list (bA, length=LA) || list (bB, length=LB)*
   *baseCD = list (bC, length=LC) || list (bD, length=LD)*
   *zetaAB = list (zA, length=LA) || list (zB, length=LB)*
   *zetaCD = list (zC, length=LC) || list (zD, length=LD)*
   *zeta = zetaAB || zetaCD*

Given *(LA, LB)* and *(bA, bB)*:

1. Compute *M*:
   *M = bA ^ LA * bB ^ LB*

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

## 2.2 Keys creation

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

2. For each $priv_n$, apply $zeta_n$ iterations of a one-way function *hashUp* with output size *bits*:
   $pub = pub_0, pub_1, pub_2, \ldots, pub_{length-1}$
   $pub_n = hashUp (priv_n, iterations=zeta_n)$

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

## 2.3 Signing

Given a *message* and a private key *priv*:

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

2. Given a counter *n = 0, 1, …*

   - Compute
     *m = hashB (n || h)*
     until a value *m* is found that satisfies
     *m < M*.

3. Represent *m* as a mix-radix number with bases *baseAB*:
   *uAB = mixradix (m, baseAB)*

4. Compute the *downs* values $dAB_n$ from each $uAB_n$ value:
   $dAB = dAB_0, dAB_1, \ldots, dAB_{LA+LB-1}$
   $dAB_n = zetaAB_n - uAB_n$

5. Compute the checksum, given by:
   $check = sum\ (dAB)$

6. Represent *check* as a mix-radix number with bases *baseCD*:
   $uCD = mixradix\ (check, baseCD)$

7. Compute the *downs* values $dCD_n$ from each $uCD_n$ value:
   $dCD = dCD_0, dCD_1, \ldots, dCD_{LC+LD-1}$
   $dCD_n = zetaCD_n - uCD_n$

8. Define the *ups* list (needed for signing) as the concatenation of the two lists *dAB* and *dCD*:
   $ups = dAB \mid\mid dCD$

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

10. 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.
   $m = hashB\ (n \mid\mid h)$

3. Check that $m < M$.

4. Represent *m* as a mix-radix number with bases *baseAB*:
   $uAB = mixradix\ (m, baseAB)$

5. Compute the *downs* values $dAB_n$ from each $uAB_n$ value:
   $dAB = dAB_0, dAB_1, \ldots, dAB_{LA + LB - 1}$
   $dAB_n = zetaAB_n - uAB_n$

6. Compute the checksum, given by:
   $check = sum\ (dAB)$

7. Represent *check* as a mix-radix number with bases *baseCD*:
   $uCD = mixradix\ (check, baseCD)$

8. Define the *ups* list (needed for verification) as the concatenation of the two lists *uAB* and *uCD*:
   $ups = uAB \mid\mid uCD$

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

10. Check that $t == pub$.

11. The signature is valid if all test (steps 3 and 10) evaluate to true, invalid otherwise.

## 2.5 Auxiliary functions

**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.
```

# 3. Table of L, z constants

The table shows computed *L, z* values for parameters *bits=256* and various lengths $L_p$.
$L = LA, LB, LC, LD$
$z = zA, zB, zC, zD$

Other constants can be derived easily from them:
$b_n = z_n + 1$
$M = bA \wedge LA * bB \wedge LB$

**(bits=256)**

| length | L | z |
|---|---|---|
| 16 | (15, 0, 1, 0) | (137270, 137269, 2059049, 2059048) |
| 20 | (8, 10, 2, 0) | (19112, 19111, 586, 585) |
| 24 | (10, 12, 2, 0) | (3183, 3182, 264, 263) |
| 28 | (12, 14, 2, 0) | (920, 919, 154, 153) |
| 32 | (16, 14, 1, 1) | (370, 369, 105, 104) |
| 40 | (26, 12, 1, 1) | (106, 105, 63, 62) |
| 48 | (17, 29, 1, 1) | (47, 46, 46, 45) |
| 56 | (40, 14, 1, 1) | (26, 25, 37, 36) |
| 64 | (21, 40, 1, 2) | (18, 17, 10, 9) |
| 80 | (2, 75, 3, 0) | (10, 9, 8, 7) |
| 96 | (82, 10, 4, 0) | (6, 5, 4, 3) |
| 112 | (20, 88, 3, 1) | (5, 4, 4, 3) |
| 128 | (25, 99, 2, 2) | (4, 3, 4, 3) |
| 160 | (25, 130, 2, 3) | (3, 2, 3, 2) |
| 192 | (120, 66, 4, 2) | (2, 1, 2, 1) |
| 224 | (67, 150, 2, 5) | (2, 1, 2, 1) |
| 256 | (12, 237, 2, 5) | (2, 1, 2, 1) |

## 4. Evaluation

To evaluate the scheme we will take into account the cost of keys creation *W*, given by the hash evaluation needed to go from the private key to the public key. The cost *W* equals the costs of signing and verifying combined, and can be computed from the equation:

$W = La * zA + LB * zB + LC * zC + LD * zD$

The following table compares the cost of keys creation for 256-bit uwots and wots+ signatures of the same size. Notice that the performance improvement tends to increase as the signatures get smaller (more compressed).

(bits=256)

| L | length (bits) | uwots | wots+ | relation |
|---|---|---|---|---|
| 20 | 5376 bits | 345.18 $ms_h$ | 655.34 $ms_h$ | 52.7% |
| 22 | 5888 bits | 143.37 $ms_h$ | 180.20 $ms_h$ | 79.6% |
| 28 | 7424 bits | 24.21 $ms_h$ | 28.64 $ms_h$ | 84.5% |
| 39 | 10240 bits | 4.57 $ms_h$ | 4.95 $ms_h$ | 92.3% |
| 55 | 14336 bits | 0.15 $ms_h$ | 0.17 $ms_h$ | 89.7% |

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 uwots family of signatures. The code can be found at:

[1] http://jotasapiens.com/