

The Walnut Digital Signature AlgorithmTM Specification

Iris Anshel, Derek Atkins, Dorian Goldfeld, and Paul Gunnells

SecureRF Corporation

100 Beard Sawmill Rd #350, Shelton, CT 06484

ianshel@securerf.com, datkins@securerf.com, dgoldfeld@securerf.com, pgunnells@securerf.com

Abstract. This document specifies the Walnut Digital Signature Algorithm (WalnutDSA), a submission to the NIST Post-Quantum Cryptography standardization process. WalnutDSA is a group-theoretic signature system based on non-abelian group theory combined with matrices, permutations, and arithmetic over finite fields. The computation imbalance has signature verification significantly more efficient than signature generation, allowing fast verification even on constrained 16- and 8-bit processors. Key and signature sizes are comparable to efficient non-quantum-resistant methods, and are much smaller than most other quantum-resistant methods.

Table of Contents

1	Introduction	2
2	Definitions	2
3	Design Rationale	3
4	Key Generation	4
5	Signature Generation	4
5.1	Message Encoding	5
5.2	Cloaking Elements	5
5.3	Rewriting	6
5.3.1	BKL Normal Form	6
5.3.2	Stochastic Rewriting	6
5.3.3	Dehornoy Reduction	8
6	Signature Verification	9
7	<i>E</i> -Multiplication	9
8	Permutations	10
9	Braids	10
10	Object Encodings	11
10.1	Public Key	12
10.2	Signature	12
11	Parameter Specifications	12
11.1	Security Level: 128	12
11.2	Security Level: 256	13
11.3	Security Level: 40 – For Testing Purposes	13
A	Stochastic Rewriting “Y” Generator Relations	14
B	Expected Computational Efficiency and Performance	16
B.1	Key-Pair Generation	16

B.2	Raw Signature Generation	17
B.3	Signature Rewriting	17
B.4	Signature Validation	19
C	Expected Security Strength	21
D	Advantages and Limitations	22
E	Known Attacks	22

1 Introduction

The Walnut Digital Signature Algorithm (WalnutDSA) was introduced in 2016 [1] based on research for a lightweight signature method useful for constrained and embedded devices. The foundation of WalnutDSA is *E*-Multiplication, a one-way function published in 2005, which combines infinite group theory in a non-abelian group, matrices, permutation, lookups, and finite field arithmetic. Specifically, the group used by WalnutDSA is the Braid group. See [1] for a full security evaluation and additional references, including a proof of security under EUF-CMA.

In WalnutDSA a private key and a signature are braids, and a public key is a pair of matrices and permutations, and an ordered list of entries in the finite field called T-values. To generate a signature one takes the message to be signed, hashes it, converts the hash output to a braid, and then builds the signature. To verify a signature one performs the same hashing and braid-converstion of the message, then two sets of *E*-Multiplication operations, one matrix multiplication, and then compares the results.

E-Multiplication is extremely lightweight, easily implemented, and runs very efficiently even in small environments. This makes signature validation very efficient even on small, 16- and 8-bit processors. In lightweight hardware an *E*-Multiplication can occur in a single clock cycle.

This document specifies how to implement WalnutDSA including key generation, signature generation, and signature validation. It also includes the necessary statements to meet the requirements of the NIST Post-Quantum Standardization Process section 2.B, modulo the attached references (e.g., [1]).

2 Definitions

- **B_N** : The braid group on N strands.
- **Braid**: A configuration of N woven strands. The braid group is denoted B_N .
- **Braid Generators**: The braid group B_N has $N - 1$ Artin generators labeled b_1, b_2, \dots, b_{N-1} which, along with their inverses (b_i^{-1}) can be used to define any braid.
- **CB Matrix**: A colored Burau matrix. c.f. [7].
- **Cloaking Element**: A braid of a special form that disappears during *E*-Multiplication. A cloaking element v is said to cloak for a permutation σ when $(m, \sigma) \star v = (m, \sigma)$.
- ***E*-Multiplication**: An action on a matrix, permutation, and braid resulting in a new matrix and permutation. The action is denoted by the symbol \star .
- **\mathbb{F}_q** : A finite field with q elements. In the case of a binary field, the polynomial elements of the field are translated directly into the binary integer representation.

- **Matrix:** An $N \times N$ grouping of elements. Herein these elements are computed over a finite field \mathbb{F}_q .
- **Permutation:** A one-to-one function from a set onto itself.
- **Purebraid:** A non-trivial braid whose corresponding permutation is trivial.
- **\mathcal{R} :** A braid rewriting function that maps a braid to another equivalent braid, used to obfuscate the structure.
- **σ_w :** A permutation of the braid word w .
- **T-values:** An array of N nonzero elements $\{\tau_1, \tau_2, \dots, \tau_N\}$ in the finite field \mathbb{F}_q .

3 Design Rationale

During the construction of WalnutDSA we encountered several decision points. The following are potential questions about WalnutDSA that may arise and the answers as to why the choices were made. Note that understanding these issues may require reading to the end and then returning to this section.

- Why are cloaking elements needed?

Cloaking elements exist to prevent WalnutDSA from being susceptible to the Conjugacy Search Problem.

The general goal was to produce a group element, using the signer’s private key and the message, so that extracting the signer’s private key would be algorithmically hard. The second goal was to use E-Multiplication for the verification step. The third goal was that the public key of the signer and some other readily available data facilitates one side of the verification. In order to bring the message into the signature, a hash function is applied and then a method for encoding the hash output into the braid group is performed. Since simply conjugating the encoded message by the signer’s private key would not be sufficient to keep the private key secure, additional braids are inserted into the conjugate to obscure the private key. In order to facilitate (efficient) verification, these additional braids must not be too long and must satisfy the cloaking property.

- Why do we set τ_1 and $\tau_2 = 1$?

In order to produce cloaking elements which are reasonably short, we need two of the T-values to be $= 1$. Significant testing showed that when a specific T-value (τ_i) is $= 1$, then the i^{th} row of the public key and verification matrices significantly duplicates the values from the row before. Additional testing demonstrates that, by choosing the first two T-values to be $= 1$, the matrix produced in the public key and verification steps of the method will have fewer repeated entries, at the expense of additional zeros in the top-right corner of the matrix. We take these zeros into account in our analysis of the security level.

- Why are hashed messages encoded into a free subgroup of the pure braid subgroup?

We need to ensure that two distinct hashes of messages encode in distinct ways, i.e., yield different braids. We do this by associating with each possible block of bits in a hash value with a positive power of a braid which is a generator of a free subgroup of the braid group. Such expressions are necessarily distinct.

- Why are $N \geq 8$ and $q \geq 32$?

We need to ensure that a brute force attack on the method is ineffective. By choosing $N \geq 8$ and $q \geq 32$, we ensure there are sufficiently many public keys, and that the underlying algorithmically hard problems are robust.

4 Key Generation

First generate the private key:

1. Choose an integer $N \geq 8$ and associated braid group B_N .
2. Choose a finite field \mathbb{F}_q of $q \geq 32$ elements.
3. Compute the value L from the security level, which determines the minimal length of certain random braid words. $L = \lceil \text{Security Level}/2 \log_2(N(N - 1)) \rceil$.
4. Compute the value ℓ from the security level, which determines the minimal length of the private key. To compute ℓ you need to solve the equation:

$$\ell = \text{Security Level} + \log_2(\ell) - \log_2(N - 1) - \log_2\left(\binom{\ell - 2 + N}{N - 1}\right)$$

Note that these four items are pre-defined by the security level parameters in Section 11 so in general nothing must be done to “choose” N , q , L , or ℓ .

5. Choose a random set of T-values $= \{\tau_1, \tau_2, \dots, \tau_N\}$, where each τ_i is an invertible element in \mathbb{F}_q ($\tau_i \neq 0, 1$), and then set $\tau_1 = \tau_2 = 1$.
6. Generate a random braid of length ℓ (see section 9). This braid must not be a purebraid, so regenerate if the permutation is trivial.
7. Freely reduce this braid (see section 5.1).
8. Generate a second random braid of length ℓ and freely reduce it. This braid must not be a purebraid, and must not have the same permutation as the first braid, so regenerate if the permutation is trivial or matches the permutation of the first braid.
9. $\text{Priv}(S)$ is the first freely reduced braid, which has permutation $\sigma_{\text{Priv}S}$, and $\text{Priv}(S')$ is the second freely reduced braid, which has permutation $\sigma_{\text{Priv}S'}$.

Next, compute the public key by E -Multiplication (\star):

1. $\text{Pub}(S) = (\text{Id}_N, \text{Id}_{S_N}) \star \text{Priv}(S)$, where Id_N is the $N \times N$ identity matrix and Id_{S_N} is the identity permutation in S_N . See section 7 for more information on E -Multiplication.
2. $\text{Pub}(S') = (\text{Id}_N, \text{Id}_{S_N}) \star \text{Priv}(S')$
3. Publish the public key with the following data:
 - N
 - q
 - T-values
 - $\text{Pub}(S)$
 - Matrix Part of $\text{Pub}(S')$

5 Signature Generation

The WalnutDSA method requires the full message to be hashed before being passed into the signature function. An appropriate hash method should be used and the output passed to the signature generator as the message \mathcal{M} . Suggested hash functions are denoted in the appropriate parameter selections (see Section 11).

To generate a signature:

1. Generate the encoded message $E(\mathcal{M})$ as per section 5.1.
2. Generate cloaking elements v_1, v_2, v_3 as defined in section 5.2, where v_1 cloaks for the identity, v_2 cloaks for $\sigma_{PrivS'}$, and v_3 cloaks for σ_{PrivS} .
3. Compute the signature $Sig = \mathcal{R}(v_3 \cdot \text{Priv}(\mathbf{S})^{-1} \cdot v_1 \cdot E(\mathcal{M}) \cdot \text{Priv}(\mathbf{S}') \cdot v_2)$, a braid rewritten as per 5.3.

5.1 Message Encoding

The original input gets hashed and the output of the hash is the input to the WalnutDSA Message Encoder. We assume the hash output \mathcal{M} is 4ℓ bits long.

The encoding method utilizes the collection of pure braid generators given by the following equations [2]:

$$\begin{aligned} x_1 &= g_{(N-1),N} = b_{N-1}^2 \\ x_2 &= g_{(N-2),N} = b_{N-1} \cdot b_{N-2}^2 \cdot b_{N-1}^{-1} \\ x_3 &= g_{(N-3),N} = b_{N-1} b_{N-2} \cdot b_{N-3}^2 \cdot b_{N-2}^{-1} b_{N-1}^{-1} \\ x_4 &= g_{(N-4),N} = b_{N-1} b_{N-2} b_{N-3} \cdot b_{N-4}^2 \cdot b_{N-3}^{-1} b_{N-2}^{-1} b_{N-1}^{-1} \\ &\vdots \\ x_{N-1} &= g_{1,N} = b_{N-1} b_{N-2} \cdots b_2 \cdot b_1^2 \cdot b_2^{-1} b_3^{-1} \cdots b_{N-1}^{-1} \end{aligned}$$

For this specification we use the generators $x_{N-1}, x_{N-3}, x_{N-5}$, and x_{N-7} , which we call g_1, g_2, g_3 , and g_4 .

The message encoder proceeds as follows:

1. Break the message \mathcal{M} into ℓ 4-bit blocks.
2. For each block:
 - (a) The low two (2) bits determine the generator g_i ($1 \leq i \leq 4$).
 - (b) The high two (2) bits determine the power $1 \leq e \leq 4$.
 - (c) Compute the braid g_i^e corresponding to this block.
 For example, the hex hash output $0x1234$ results in the output $g_2 \ g_3 \ g_4 \ g_1^2$
3. The encoded message $E(\mathcal{M})$ is the freely reduced product of these ℓ block results.

Free reduction is a rewriting process that removes certain pairs of elements in the braid – namely, a generator b_i followed immediately by its inverse b_i^{-1} , or an inverse generator b_i^{-1} followed by the generator b_i . Any such pair of consecutive elements may be erased from the braid. For example, the free reduction of the braid $b_1 b_2 b_2^{-1} b_3$ is $b_1 b_3$.

5.2 Cloaking Elements

A cloaking element is a braid of a special form that disappears during E -Multiplication (see section 7). A cloaking element v is said to cloak for a permutation σ when $(m, \sigma) \star v = (m, \sigma)$.

To generate a cloaking element that cloaks permutation σ :

1. Pick a random integer $2 \leq i \leq N - 1$.

2. Compute the permutation preimages (a, b) for 1 and 2 in σ , i.e., a is the value σ takes to 1, and b is the value σ takes to 2: $a = \sigma^{-1}(1)$, $b = \sigma^{-1}(2)$.
3. Choose a random permutation σ_w of high order that moves $i \rightarrow a$, $i + 1 \rightarrow b$ (see section 8).
4. Generate a random braid using permutation σ_w (see section 9) and invert it (so the result has permutation σ_w), calling the result w .
5. Extend w with L pure braids.
6. Compute the cloaking element $v = w \cdot b_i^2 \cdot w^{-1}$.

5.3 Rewriting

The signature braid must be rewritten to hide the form and protect the private key. There are an infinite number of equivalent braids which means it is computationally infeasible to determine the original from of the rewritten version. We specify two methods to rewrite the signature (BKL Normal Form and Stochastic Rewriting), plus a third option that may be used to reduce the final length (Dehornoy Reduction).

5.3.1 BKL Normal Form

Birman–Ko–Lee (BKL) Normal Form was introduced in 1998 [3] as a canonical form for a braid. Every braid can be converted to BKL Normal Form, and every equivalent braid will result in the same BKL output. For example, the braids $b_1 b_2 b_1$ and $b_2 b_1 b_2$ would result in the same output after running through BKL.

Please reference Section 4 of [3] for the algorithm description.

5.3.2 Stochastic Rewriting

Stochastic Rewriting is a new method which is useful for smaller processors because it just involves random rewriting from lookup tables. The process is:

1. Freely reduce the braid if it has not already been reduced.
2. Convert the braid to “Y generators” and freely reduce the result.
3. Partition the braid into chunks of random sizes between 5-10 generators each¹:
 - (a) Generate random numbers between the minimum (5) and the maximum (10) inclusive, and subtract each from the initial total until the running drops below the minimum.
 - (b) If the running total is zero then the partition has been found.
 - (c) Otherwise (running total is not zero), jump back to the position in the list where the running total was greater than minimum * maximum (50) and set the running total to this value.
 - (d) Repeat until the running total becomes zero.
4. For each partition, choose a random offset into the partition (from the first to second-last).
5. Take the offset and offset+1 generators and look up a relation in the relation table (see Appendix A).

¹ Note that there are 2^{W-1} possible partitions of a word of length W if partitions could be of any size. Limiting to blocks of size 5-10 reduces that number. Still, reversing the process is hard.

6. Replace those two generators with the relation (if one is found) using the Pair Replacement method described below; otherwise, do nothing for this partition.
7. Once you reach the last partition, freely reduce and then return to step 3 in order to repeat the process 3 times.
8. After the third repetition, convert back to Artin generators and freely reduce.

Convert Artin to Y Generators

Use the following process to convert a single Artin generator $b_k \in B_N$ to Y generators using the partition p of $N - 1$. If the Artin generator is an inverse, then invert the Y result. Converting a full word just involves iteration of this method. Note that the partition here is a static partition of $N - 1$.

```
// Create an array the same length as the number of partitions of N-1
r := array[length_of(p)+1]

// Initialize this array with the sum of the partitions
r[1] = 1
for i in (2..length_of(p)+1):
    r[i] = r[i-1] + p[i-1]

// Determine which partition contains k
j = 1
while(r[j] <= k):
    j++
j--;

// Build the response
u = k - r[j]
if (u < p[j] - 1):
    answer = {{r[j]+u, 1}, {r[j]+u+1, -1}}
else:
    answer = {{r[j]+u, 1}}

return answer
```

Convert Y to Artin Generators

Use the following process to convert a single Y generator $y_k \in B_N$ to Artin generators using the partition p of $N - 1$. If the Y generator is an inverse, then invert the Artin result. Converting a full word just involves iteration of this method. Note that the partition here is a static partition of $N - 1$.

```
// Create an array the same length as the number of partitions of N-1
r := array[length_of(p)+1]

// Initialize this array with the sum of the partitions
r[1] = 1
```

```

for i in (2..length_of(p)+1):
    r[i] = r[i-1] + p[i-1]

// Determine which partition contains k
j = 1
while(r[j] <= k):
    j++
j--;

// Build the response
u = k - r[j]
answer = {}
for i in (0..p[j]-u-1):
    answer = answer + {r[j]+u+i, 1}

return answer

```

Integer Partitions

A partition of an integer is an ordered list of integers that sum to the desired total. For example, the integer 20 can be partitioned into {10,10}, or {5,5,5,5}, or {7,8,5}, or {7,5,8}, or any other random split.

Pair Replacement

The partitioning of $N - 1$ not only defines the Artin to Y generator mappings but also the set of relations between pairs of generators. This list of relations is enumerated in Appendix A. Given a Y-word of length two, the process searches through the list of relations until a match is found (i.e, it finds a relation that contains the length-two word). Note that not all pairs of generators have relations, so there may not be a match.

Once a relation is found, the replacement is made by replacing the original pair with the inverse of the surrounding relation by inverting the part of the relation to the left of the subword, and then the part to the right of the subword. For example, if one is searching for the subword {1,1}{2,1} and the relation that was found is {3,1}{2,1}{1,1}{2,1}{3,1}. The left part is {3,1}{2,1} and the right part is {3,1}. Once you invert and concatenate these parts, then the replacement would be {2,-1}{3,-1}{3,-1}.

Note that if there are multiple matches for the pair within the relation then randomly choose one of the matches.

5.3.3 Dehornoy Reduction

Dehornoy Reduction is a method to reduce the size of a braid by finding and removing complex cancelations beyond single free reduction [4]. While solving the shortest word problem in the braid group is known to be NP-Hard, Dehornoy is the best-known method to reduce a braid to a minimal length. Applying Dehornoy is recommended; however, it may be applied “later”. For example, a lightweight processor may generate a signature and use Stochastic Rewriting,

and then send that (long) signature to another, more powerful device, which can then run Dehornoy.

Please reference Section 4 of [4] for the algorithm description.

6 Signature Verification

Signature verification depends on first hashing the input using the same hash method as the signature generation to generate the hash output \mathcal{M} . Then to verify the signature:

1. Compute $E(\mathcal{M})$ as per section 5.1.
2. Evaluate $(M_1, \sigma_1) = (\text{Id}_N, \text{Id}_{S_N}) \star E(\mathcal{M})$, where Id_N is the $N \times N$ identity matrix and Id_{S_N} is the identity permutation in S_N .
3. Evaluate $(M_2, \sigma_2) = \text{Pub}(S) \star \text{Sig}$.
4. Compute the matrix multiplication $M_3 = M_1 \cdot \text{MatrixPart}(\text{Pub}(S'))$.
5. Compare M_2 and M_3 for equality. If $M_2 = M_3$, then the signature is valid.

7 *E*-Multiplication

The one-way function *E*-Multiplication is an action that starts with a matrix and permutation, a braid, and results in a new matrix and permutation. *E*-Multiplication is iterative, and by definition is applied one braid generator at a time. One can find closed formula for applying certain longer braid words.

The best way to explain the process is via pseudo-code. To compute a single *E*-Multiplication starting with a matrix m , permutation p , and braid generator b_i^e :

```
// compute the multiplication values for this generator based
// on the T-values, strand, and whether the generator is inverted
if e == 1:
    a = T[p[i]]
    b = -a
    c = 1
else: // e == -1
    a = 1
    b = -Inverse(T[p[i+1]])
    c = -b

// iterate down columns and matrix-multiply each value
if i != 0:
    for j in (1..N):
        m[j][i-1] += m[j][i] * a

for j in (1..N):
    m[j][i+1] += m[j][i] * c

for j in (1..N):
```

```

m[j][i] *= b

// swap permutation based on the generator
temp = p[i]
p[i] = p[i+1]
p[i+1] = temp

```

To compute the E -Multiplication of a longer braid, one just iterates this process over the whole braid, reading from left to right.

8 Permutations

To generate a random permutation use the Fisher-Yates Shuffle [5]:

1. Start with the identity permutation of n elements.
2. Start with the last (1-indexed) offset, $i = n$.
3. Choose a random number $1 \leq j \leq i$.
4. Swap permutation elements i and j .
5. Iterate i down to 1.

If the desired permutation has additional constraints, those constraints can be applied after this process is complete. For example, if one needs to move $i \rightarrow a$, then one takes a randomly constructed permutation and modifies it by the following:

1. Find the permutation preimage of a . This is the offset o where the permutation value is a .
2. Swap the entries at o and i .

Note that all offsets and values are 1-indexed in this definition.

9 Braids

To generate a random braid word of length l :

1. Choose a random braid generator b_i , where $1 \leq i < N$.
2. Choose a random power, $\epsilon = \{-1, 1\}$.
3. Append b_i^ϵ to the braid word.
4. Iterate l times.
5. Freely reduce the result.

To generate a random braid $b(\sigma)$ from permutation σ :

1. Convert the permutation σ to a product of transpositions $t_1 \cdots t_r$:
 - (a) First write σ as a product of disjoint cycles $C_1 \cdots C_s$ where the last element of each C_i is the smallest number in the cycle.
 - (b) Order the cycles such that the last element of each C_i is in ascending order.
 - (c) Convert each cycle to a product of transpositions: if $C_i = (a_1, \dots, a_k)$, then $C_i = (a_1, a_2)(a_1, a_3) \cdots (a_1, a_k)$.

- (d) Replace each C_i with its corresponding product of transpositions and flatten the list.
- 2. For each transposition t_i , generate a random braid $b(t_i)$ that produces it:
 - (a) Find the smallest element m and largest element M exchanged by the transposition t , i.e., $t = (m, M)$.
 - (b) Set $b(t_i)$ to be the identity braid.
 - (c) For $k = m$ to $M - 1$, replace $b(t_i)$ with $b(t_i) \cdot b_k^\epsilon$, where $\epsilon = \{-1, 1\}$ is randomly chosen.
 - (d) For $k = 2$ to $M - m$, replace $b(t_i)$ with $b(t_i) \cdot b_{M-k}^\epsilon$, where again $\epsilon = \{-1, 1\}$ is randomly chosen.
- 3. The result² $b(\sigma)$ is the product $b(t_1) \cdots b(t_r)$.

To meet specific security constraints, the braid can be augmented with pure braids. Specifically, it is the freely reduced product of L pure braid generators. The pure braid subgroup of B_N is generated [6] by the set of $(N)(N - 1)/2$ braids given by:

$$g_{i,j} = b_{j-1} b_{j-2} \cdots b_{i+1} \cdot b_i^2 \cdot b_{i+1}^{-1} \cdots b_{j-2}^{-1} b_{j-1}^{-1}, \quad 1 \leq i < j \leq N.$$

To create a pure braid generator of B_N :

- 1. Choose random numbers $i, j: 1 \leq i < j \leq N$.
- 2. Choose a random exponent $\epsilon = \{-1, 1\}$.
- 3. Iterate $0 \leq k < j - i - 1$ and append b_{j-k-1} .
- 4. Append $b_i^{2\epsilon}$.
- 5. Iterate $0 \leq k < j - i - 1$ and append b_{i+k+1}^{-1} .

10 Object Encodings

Throughout this document, indices are often 1-indexed. For example, in B_N the generators are labeled b_1, b_2, \dots . However, computers are better with 0-indexed numbers, arrays, and matrices, so the encodings are 0-indexed.

For all encodings, multi-byte numbers are encoded in network byte order (i.e., most significant byte first). For example, the decimal number 255 is encoded in hex as 00 FF, decimal 256 as 01 00.

Larger data objects like matrices, permutations, and braids are “bit packed” to reduce the effective transmission size. Bit packing also uses most-significant-bit first. When packing a matrix, the entries are encoded from 0 to $q - 1$ and packed across each row sequentially. The permutation is packed as a series of entries from 0 to $N - 1$.

Braids get encoded first with a 2-byte length (which is the number of generators)³, and then each generator is encoded with one bit for the sign and additional bits for the strand. For example, in B_8 packing each braid generator requires 4 bits. In B_8 , encoding the braid $b_1 b_2^{-1} b_3 b_4 b_5^{-1} b_6 b_7^{-1}$ would result in the hex 00 07 09 23 C5 E0.

² Note that this process generates a braid that has permutation σ^{-1} which is why it is inverted in Section 5.2.

³ In all testing to date, signature braids have never exceeded 10,000 generators.

10.1 Public Key

The public key contains the following data:

- **N**: an 8-bit unsigned integer.
- **q**: a 16-bit unsigned integer.
- **T-values**: a packed array of N entries in \mathbb{F}_q . This results in $N \log_2(q)$ bits, which gets rounded up to the nearest byte (padded with 0-7 bits of zeros).
- **Pub(S) Matrix**: a packed matrix of $N \times (N - 1) + 1$ entries in \mathbb{F}_q . We know the last row of the matrix is always 0, except for the last entry of the last row, so those $N - 1$ entries are elided from the packing. This results in $(N^2 - N + 1) \log_2(q)$ bits which gets rounded up to the nearest byte (padded with 0-7 bits of zeros).
- **Pub(S) Permutation**: a packed array of N entries from 0 to $N - 1$. This results in $N \log_2(N)$ bits, which gets rounded up to the nearest byte (padded with 0-7 bits of zeros).
- **Pub(S') Matrix**: a packed matrix of $N \times (N - 1) + 1$ entries in \mathbb{F}_q . We know the last row of the matrix is always 0, except for the last entry of the last row, so those $N - 1$ entries are elided from the packing. This results in $(N^2 - N + 1) \log_2(q)$ bits which gets rounded up to the nearest byte (padded with 0-7 bits of zeros).

10.2 Signature

A signature is just a braid, so it is encoded as a single packed braid as detailed at the start of this section. It has a 2-byte (16-bit) integer length (the number of generators) followed by the packed list of generators. Because each generator encodes into 4 bits, you can fit two generators into every byte. If you have an odd number of generators then the final byte is padded with zeros.

It does not matter which reduction method is used; in all cases the signature is converted to and encoded in Artin generators. Moreover, the 2-byte length field is sufficient because in all cases the maximum length seen experimentally is well below 65,000 generators. A long signature could be the result of an attempted attack and must be considered invalid.

11 Parameter Specifications

11.1 Security Level: 128

For a classical security level of 2^{128} (which, subject to Grover, results in a quantum-safe security level of 2^{64}), use the following parameters:

- $N = 8$
- $q = 32$ (using polynomial $x^5 + x^2 + 1$)
- $L = 15$
- $\ell = 132$
- Hash function: SHA2-256

This results in a 256-bit message size, at least 2^{200} possible public keys that would need to be searched, as well as at least 2^{128} possible secret keys and cloaking elements. The public key is 664 bits (including the N/q values). The private key is variable length and has a maximum

length of 1056 bits (not including any markers as to N , individual braid lengths, or the security level).

Signatures are variable length, and the actual resulting length also depends on which rewriting method gets used. Experimentally we can determine the expected minimum, maximum, and average lengths (see Table 1).

Rewriting Method	Minimum	Mean	Maximum
BKL + Dehornoy	3080	5172.5	7704
Stochastic + Dehornoy	3056	5134.6	7616
Stochastic w/o Dehornoy	8944	11331.6	13968

Table 1: Experimentally determined 128-bit signature lengths (in bits)

11.2 Security Level: 256

For a classical security level of 2^{256} (which, subject to Grover, results in a quantum-safe security level of 2^{128}), use the following parameters:

- $N = 8$
- $q = 256$ (using polynomial $x^8 + x^4 + x^3 + x + 1$)
- $L = 30$
- $\ell = 287$
- Hash function: SHA2-512

This results in a 512-bit message size, at least 2^{320} possible public keys that would need to be searched, as well as at least 2^{256} possible secret keys and cloaking elements. The public key is 1024 bits (including the N/q values). The private key is variable length with a maximum of 2296 bits (not including any markers as to N , individual braid lengths, or the security level).

Signatures are variable length, and the actual resulting length also depends on which rewriting method gets used. Experimentally we can determine the expected minimum, maximum, and average lengths (see Table 2).

Rewriting Method	Minimum	Mean	Maximum
BKL + Dehornoy	6784	9981.6	13880
Stochastic + Dehornoy	6768	9932.4	13552
Stochastic w/o Dehornoy	17552	21556.6	25240

Table 2: Experimentally determined 256-bit signature lengths (in bits)

11.3 Security Level: 40 – For Testing Purposes

In order to test an insecure version of WalnutDSA, we suggest a smaller version at a classical security level of 2^{40} by using the following parameters:

- $N = 8$
- $q = 16$
- $L = 4$
- $\ell = 25$
- Hash function: SHA1

This results in a 160-bit message size, at least 2^{160} possible public keys that would need to be searched, as well as at least 2^{40} possible secret keys and cloaking elements. The public key is 544 bits (including the N/q values). The private key is variable length with a maximum of 200 bits (not including any markers as to N or the security level).

References

1. Anshel, I.; Atkins, D.; Goldfeld, D.; Gunnells, P. E., *WalnutDSATM: A Lightweight Quantum Resistant Digital Signature Algorithm*, <https://eprint.iacr.org/2017/058.pdf>.
2. Birman, J., *Braids, Links and Mapping Class Groups*, Annals of Mathematics Studies, Princeton University Press, 1974.
3. Birman, J.; Ko, K. H.; Lee, S. J., *A new approach to the word and conjugacy problems in the braid groups*, Adv. Math. 139 (1998), no. 2, 322–353, <http://www.math.columbia.edu/~jb/bkl-newpres.pdf>.
4. Dehornoy, P. *A fast method for comparing braids*, Adv. Math. 125 (1997), no. 2, 200–235, <http://www.math.unicaen.fr/~dehornoy/Papers/Dfo.pdf>.
5. Fisher, R.; Yates, F., *Fisher–Yates Shuffle*, http://en.wikipedia.org/wiki/Fisher-Yates_shuffle.
6. Hansen, V. L., *Braids and coverings: selected topics*, With appendices by Lars Gæde and Hugh R. Morton, London Mathematical Society Student Texts, 18, Cambridge University Press, Cambridge, (1989).
7. Morton, H. R., *The multivariable Alexander polynomial for a closed braid*, *Low-dimensional topology*, (Funchal, 1998), 167–172, Contemp. Math., 233, Amer. Math. Soc., Providence, RI, 1999.

A Stochastic Rewriting “Y” Generator Relations

The following is a list of 1-indexed Y relations in B_8 with partition {4,3} for use in the Stochastic Rewriting process. This list is created by enumerating all braid relations available in Artin generators (e.g., $b_1 b_2 b_1 = b_2 b_1 b_2$, etc, and $b_1 b_3 = b_3 b_1$, etc) and converting them to Y generators, and also the additional relations that are available due to the change over to Y generators. Note that not every possible pair of $y_i y_j$ is in the list. This list is then sorted by length, which results in the following list:

```
{4,1}{7,1}{4,-1}{7,-1}
{7,-1}{4,1}{7,1}{4,-1}
{7,1}{4,1}{7,-1}{4,-1}
{2,1}{1,1}{4,1}{1,-1}{1,-1}
{3,1}{2,1}{4,1}{2,-1}{2,-1}
{6,1}{5,1}{7,1}{5,-1}{5,-1}
{7,1}{6,1}{7,1}{6,-1}{6,-1}
{1,-1}{2,1}{1,1}{4,1}{1,-1}
{2,-1}{3,1}{2,1}{4,1}{2,-1}
{5,-1}{6,1}{5,1}{7,1}{5,-1}
{6,-1}{7,1}{6,1}{7,1}{6,-1}
{1,1}{1,1}{4,-1}{1,-1}{2,-1}
```

$\{2,1\}\{2,1\}\{4,-1\}\{2,-1\}\{3,-1\}$
 $\{5,1\}\{5,1\}\{7,-1\}\{5,-1\}\{6,-1\}$
 $\{6,1\}\{6,1\}\{7,-1\}\{6,-1\}\{7,-1\}$
 $\{2,-1\}\{1,1\}\{1,1\}\{4,-1\}\{1,-1\}$
 $\{3,-1\}\{2,1\}\{2,1\}\{4,-1\}\{2,-1\}$
 $\{6,-1\}\{5,1\}\{5,1\}\{7,-1\}\{5,-1\}$
 $\{7,-1\}\{6,1\}\{6,1\}\{7,-1\}\{6,-1\}$
 $\{3,1\}\{3,1\}\{4,-1\}\{3,-1\}\{4,-1\}$
 $\{6,1\}\{6,1\}\{7,-1\}\{6,-1\}\{7,-1\}$
 $\{4,-1\}\{3,1\}\{3,1\}\{4,-1\}\{3,-1\}$
 $\{7,-1\}\{6,1\}\{6,1\}\{7,-1\}\{6,-1\}$
 $\{4,1\}\{3,1\}\{4,1\}\{3,-1\}\{3,-1\}$
 $\{7,1\}\{6,1\}\{7,1\}\{6,-1\}\{6,-1\}$
 $\{1,1\}\{2,-1\}\{4,1\}\{2,1\}\{1,-1\}\{4,-1\}$
 $\{2,1\}\{3,-1\}\{4,1\}\{3,1\}\{2,-1\}\{4,-1\}$
 $\{4,1\}\{6,1\}\{7,-1\}\{4,-1\}\{7,1\}\{6,-1\}$
 $\{1,1\}\{2,-1\}\{7,1\}\{2,1\}\{1,-1\}\{7,-1\}$
 $\{2,1\}\{3,-1\}\{7,1\}\{3,1\}\{2,-1\}\{7,-1\}$
 $\{3,1\}\{4,-1\}\{7,1\}\{4,1\}\{3,-1\}\{7,-1\}$
 $\{5,1\}\{6,-1\}\{7,1\}\{6,1\}\{5,-1\}\{7,-1\}$
 $\{4,-1\}\{1,1\}\{2,-1\}\{4,1\}\{2,1\}\{1,-1\}$
 $\{4,-1\}\{2,1\}\{3,-1\}\{4,1\}\{3,1\}\{2,-1\}$
 $\{6,-1\}\{4,1\}\{6,1\}\{7,-1\}\{4,-1\}\{7,1\}$
 $\{7,-1\}\{1,1\}\{2,-1\}\{7,1\}\{2,1\}\{1,-1\}$
 $\{7,-1\}\{2,1\}\{3,-1\}\{7,1\}\{3,1\}\{2,-1\}$
 $\{7,-1\}\{3,1\}\{4,-1\}\{7,1\}\{4,1\}\{3,-1\}$
 $\{7,-1\}\{5,1\}\{6,-1\}\{7,1\}\{6,1\}\{5,-1\}$
 $\{4,1\}\{1,1\}\{2,-1\}\{4,-1\}\{2,1\}\{1,-1\}$
 $\{4,1\}\{2,1\}\{3,-1\}\{4,-1\}\{3,1\}\{2,-1\}$
 $\{6,1\}\{7,-1\}\{4,1\}\{7,1\}\{6,-1\}\{4,-1\}$
 $\{7,1\}\{1,1\}\{2,-1\}\{7,-1\}\{2,1\}\{1,-1\}$
 $\{7,1\}\{2,1\}\{3,-1\}\{7,-1\}\{3,1\}\{2,-1\}$
 $\{7,1\}\{3,1\}\{4,-1\}\{7,-1\}\{4,1\}\{3,-1\}$
 $\{7,1\}\{5,1\}\{6,-1\}\{7,-1\}\{6,1\}\{5,-1\}$
 $\{1,1\}\{3,-1\}\{1,1\}\{2,-1\}\{3,1\}\{1,-1\}\{3,1\}\{2,-1\}$
 $\{2,1\}\{4,-1\}\{2,1\}\{3,-1\}\{4,1\}\{2,-1\}\{4,1\}\{3,-1\}$
 $\{5,1\}\{7,-1\}\{5,1\}\{6,-1\}\{7,1\}\{5,-1\}\{7,1\}\{6,-1\}$
 $\{2,-1\}\{1,1\}\{3,-1\}\{1,1\}\{2,-1\}\{3,1\}\{1,-1\}\{3,1\}$
 $\{3,-1\}\{2,1\}\{4,-1\}\{2,1\}\{3,-1\}\{4,1\}\{2,-1\}\{4,1\}$
 $\{6,-1\}\{5,1\}\{7,-1\}\{5,1\}\{6,-1\}\{7,1\}\{5,-1\}\{7,1\}$
 $\{2,1\}\{3,-1\}\{1,1\}\{3,-1\}\{2,1\}\{1,-1\}\{3,1\}\{1,-1\}$
 $\{3,1\}\{4,-1\}\{2,1\}\{4,-1\}\{3,1\}\{2,-1\}\{4,1\}\{2,-1\}$
 $\{6,1\}\{7,-1\}\{5,1\}\{7,-1\}\{6,1\}\{5,-1\}\{7,1\}\{5,-1\}$
 $\{1,1\}\{2,-1\}\{3,1\}\{4,-1\}\{2,1\}\{1,-1\}\{4,1\}\{3,-1\}$
 $\{1,1\}\{2,-1\}\{5,1\}\{6,-1\}\{2,1\}\{1,-1\}\{6,1\}\{5,-1\}$

```

{2,1}{3,-1}{5,1}{6,-1}{3,1}{2,-1}{6,1}{5,-1}
{3,1}{4,-1}{5,1}{6,-1}{4,1}{3,-1}{6,1}{5,-1}
{1,1}{2,-1}{6,1}{7,-1}{2,1}{1,-1}{7,1}{6,-1}
{2,1}{3,-1}{6,1}{7,-1}{3,1}{2,-1}{7,1}{6,-1}
{3,1}{4,-1}{6,1}{7,-1}{4,1}{3,-1}{7,1}{6,-1}
{3,-1}{1,1}{2,-1}{3,1}{4,-1}{2,1}{1,-1}{4,1}
{5,-1}{1,1}{2,-1}{5,1}{6,-1}{2,1}{1,-1}{6,1}
{5,-1}{2,1}{3,-1}{5,1}{6,-1}{3,1}{2,-1}{6,1}
{5,-1}{3,1}{4,-1}{5,1}{6,-1}{4,1}{3,-1}{6,1}
{6,-1}{1,1}{2,-1}{6,1}{7,-1}{2,1}{1,-1}{7,1}
{6,-1}{2,1}{3,-1}{6,1}{7,-1}{3,1}{2,-1}{7,1}
{6,-1}{3,1}{4,-1}{6,1}{7,-1}{4,1}{3,-1}{7,1}
{3,1}{4,-1}{1,1}{2,-1}{4,1}{3,-1}{2,1}{1,-1}
{5,1}{6,-1}{1,1}{2,-1}{6,1}{5,-1}{2,1}{1,-1}
{5,1}{6,-1}{2,1}{3,-1}{6,1}{5,-1}{3,1}{2,-1}
{5,1}{6,-1}{3,1}{4,-1}{6,1}{5,-1}{4,1}{3,-1}
{6,1}{7,-1}{1,1}{2,-1}{7,1}{6,-1}{2,1}{1,-1}
{6,1}{7,-1}{2,1}{3,-1}{7,1}{6,-1}{3,1}{2,-1}
{6,1}{7,-1}{3,1}{4,-1}{7,1}{6,-1}{4,1}{3,-1}
{4,1}{5,1}{6,-1}{4,1}{6,1}{5,-1}{4,-1}{6,1}{5,-1}
{5,-1}{4,1}{5,1}{6,-1}{4,1}{6,1}{5,-1}{4,-1}{6,1}
{5,1}{6,-1}{4,1}{5,1}{6,-1}{4,-1}{6,1}{5,-1}{4,-1}

```

B Expected Computational Efficiency and Performance

When analyzing the computational efficiency of WalnutDSA, one must look at four distinct processes: key-pair generation, raw signature generation, signature rewriting, and signature validation.

A note on notation: when declaring the expected computation efficiency of the various subprocesses of WalnutDSA, the basis of the order is explicitly used when available. For example, using “ $\mathcal{O}(N)$ ” implies a linear operation in N , which is the number of strands of the braid in B_N , whereas using “ $\mathcal{O}(n)$ ” is a generic linear operation.

Performance was tested on a Linux server configured with 8 cores of Intel Xeon X5355 at 2.66GHz running at 2660237000 cycles per second and 32 GB RAM. The test code was compiled using: `gcc -march=native -mtune=native -O3 -fomit-frame-pointer -fwrapv -msse2`.

B.1 Key-Pair Generation

Key-pair generation is straightforward. The private key is purely a randomly generated braid, which takes $\mathcal{O}(\ell)$ operations to create and freely reduce. Generating the T-values is also an $\mathcal{O}(N)$ operation (but N is small). Finally, computing the public key via E -Multiplication requires N multiplies and $2N$ additions repeated ℓ times, so it is still $\mathcal{O}(N\ell)$ (remembering that N is fixed, so really $\mathcal{O}(\ell)$). We expect this to take fractions of a millisecond on the target

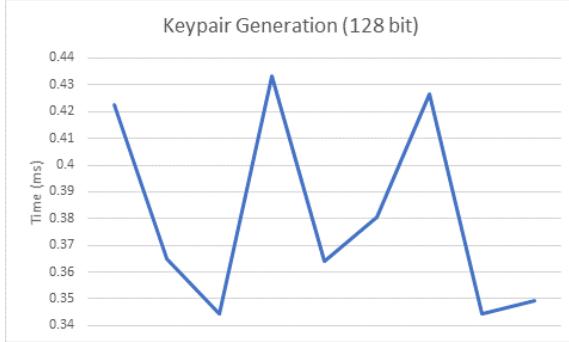


Fig. 1: Timing for generating 128-bit keys

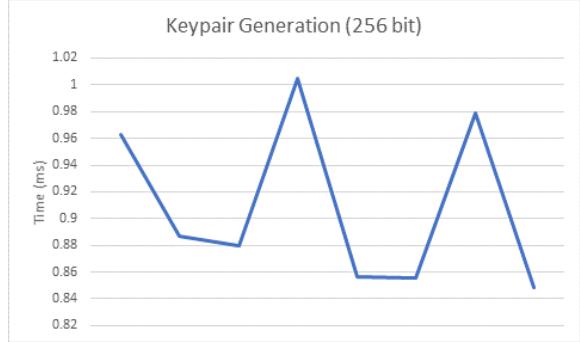


Fig. 2: Timing for generating 256-bit keys

platform. We also expect this function to be fast even on small processors because it is purely a function of the speed of the random number generator.

As shown in Figures 1 and 2, key generation on our test platform takes between 0.34-0.43ms for a 128-bit key and 0.85-1ms for a 256-bit key. This equates to 2325-2941 and 1000-1176 keys generated per second.

B.2 Raw Signature Generation

Raw signature generation is the process of taking the hash of the input message, converting the hash to a braid, generating cloaking elements, and putting the “raw” signature together prior to rewriting. We separate this from the rewriting portion because there are multiple rewriting methods proposed with different performance profiles.

Hashing the message is an $\mathcal{O}(n)$ operation in the length of the input message and is out of the control of WalnutDSA. We expect a good implementation of SHA2-256 or SHA2-512 to behave appropriately.

After hashing, we convert the hash output to a braid (this is an $\mathcal{O}(n)$ operation in the size of the hash output), and generate the three sets of cloaking elements (each an $\mathcal{O}(L)$ operation). Finally, we invert the private key ($\mathcal{O}(\ell)$) and put it all together ($\mathcal{O}(n)$). Of all these operations, the hash function is the most computationally intensive. The rest of the operations are purely limited on the speed of the random number generator. We expect this operation to be fast even on tiny devices.

B.3 Signature Rewriting

Rewriting the signature is the most computationally intensive operation in WalnutDSA, although it is required for signature security. There are three rewriting options:

1. BKL + Dehornoy
2. Stochastic Rewriting
3. Stochastic Rewriting + Dehornoy

The BKL algorithm, which outputs the canonical form of any braid, runs in $\mathcal{O}(n^2)$ time in the length of the input braid. BKL will convert any equivalent braid into the exact same

output braid, making it easy to detect “sameness.” Of course the canonical form of a braid is often much longer than the original.

Enter Dehornoy, which takes a braid and shortens it by finding ways to manipulate the braid to remove inverses, even if they are not adjacent. The Dehornoy algorithm also runs in $\mathcal{O}(n^2)$ time in the length of the input braid.

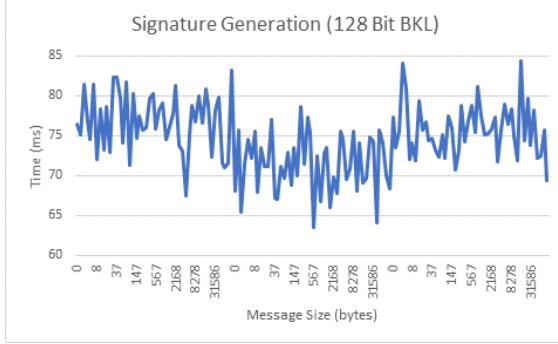


Fig. 3: SUPERCOP output for generating 128-bit signatures with BKL and Dehornoy

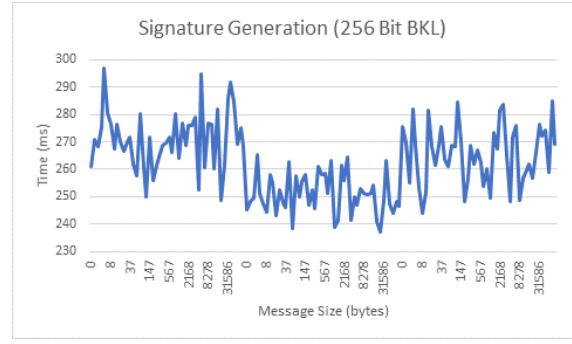


Fig. 4: SUPERCOP output for generating 256-bit signatures with BKL and Dehornoy

However, both BKL and Dehnory run in statistical time, not fixed time. Depending on the inputs they can complete very quickly or run somewhat longer. Moreover, the output is variable in length based on the inputs, which implies that WalnutDSA signatures are not constant length.

Running our implementation through SUPERCOP, we generated three sets of keys and then for each key ran 32 runs for each of 48 different message sizes (see Figure 3). At 128-bit security the signature generation took between 64-84ms. The variabiliy in execution time is due to the varying lengths of signatures across different inputs, how the hash output gets converted into varying lengths of braids, and how that interacts with the random cloaking elements.

For 256-bit security (see Figure 4), the execution time increased to 238-295ms.

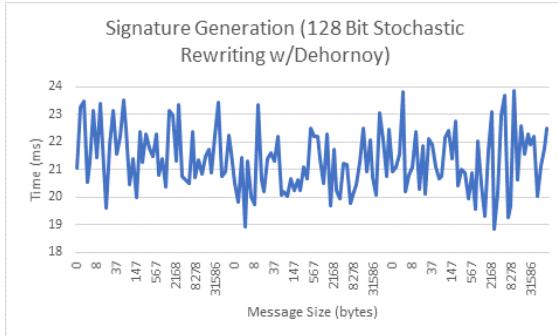


Fig. 5: SUPERCOP output for generating 128-bit signatures with Stochastic Rewriting and Dehornoy

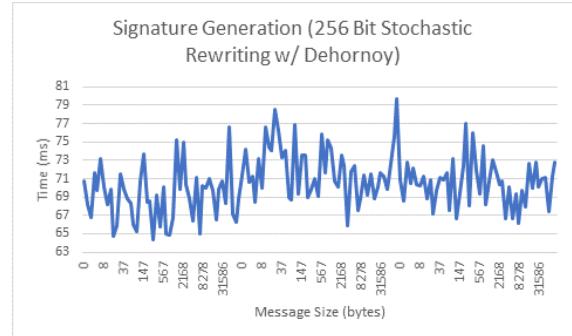


Fig. 6: SUPERCOP output for generating 256-bit signatures with Stochastic Rewriting and Dehornoy

The Stochastic Rewriting method is a mostly-linear operation that randomly replaces sections of a braid using a known set of braid relations. Its running time is slightly greater than linear, because the length of the braid increases on every round. The exact complexity is greater than $\mathcal{O}(n)$ but less than $\mathcal{O}(n \log(n))$.

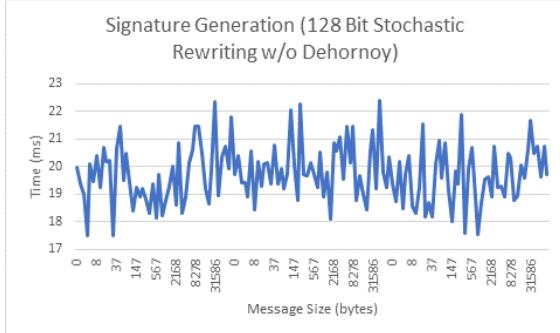


Fig. 7: SUPERCOP output for generating 128-bit signatures with Stochastic Rewriting without Dehornoy

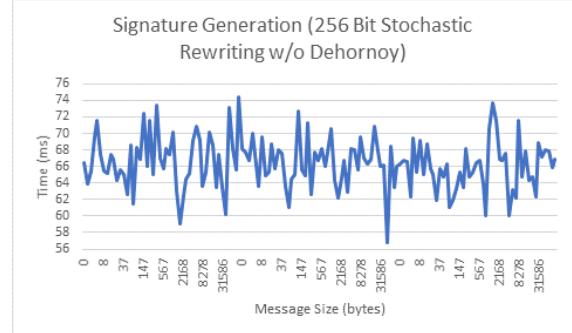


Fig. 8: SUPERCOP output for generating 256-bit signatures with Stochastic Rewriting without Dehornoy

As shown in Figures 5 and 6, replacing BKL with Stochastic Rewriting provides a significant speed increase. With the three keys randomly chosen at the 128-bit security level, signatures generated in 19-24ms.

At the 256-bit security level speed is also increased. Those keys signed messages in 64-79ms. Using Stochastic Rewriting shows a 3-4x speed improvement over BKL.

Moreover, it's likely that Stochastic Rewriting could be implemented on an embedded device. However in this case it's more likely that the embedded device would only run Stochastic Rewriting, then send the signature over to a more powerful device (trading of signature size and transmission time for computation capability). The larger device could run Dehornoy and reduce the signature for storage.

When you remove Dehornoy, our test system was able to generate a 128-bit signature in 18-22ms (see Figure 7), and a 256-bit signature in 60-74ms (see Figure 8). The complexity reduction is such that an embedded device may be sufficient; however, the resulting signature is longer.

B.4 Signature Validation

Validating a signature requires hashing the message, converting the hash output to a braid ($\mathcal{O}(n)$), two sets of E -Multiplication ($\mathcal{O}(n)$ in the length of the signature and the length of the converted hash output), one matrix multiplication ($\mathcal{O}(N^3)$), and one matrix comparison ($\mathcal{O}(N^2)$).

Figure 9 shows clearly that for smaller messages the mathematical computation dominates, but the hash computation starts to dominate once input messages reach about 8000 bytes. Specifically, looking at Table 3, it appears that the run time starts to increase once messages reach somewhere between 1500-4000 bytes, and the hash function dominates, more than doubling the execution time, between 6000-10000 bytes. A significantly optimized hash implementation is clearly a requirement.

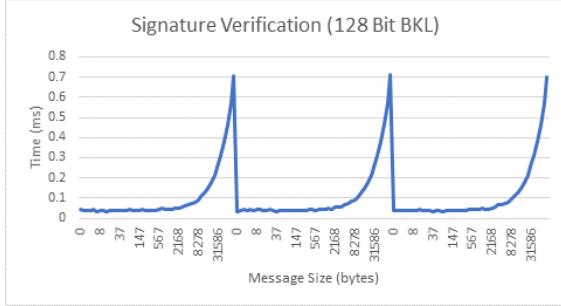


Fig. 9: SUPERCOP output for verifying 128-bit signatures with BKL and Dehornoy

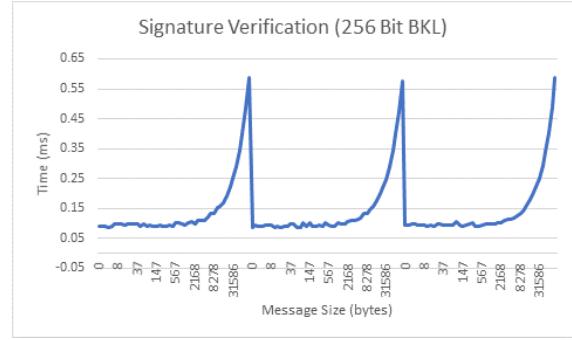


Fig. 10: SUPERCOP output for verifying 256-bit signatures with BKL and Dehornoy

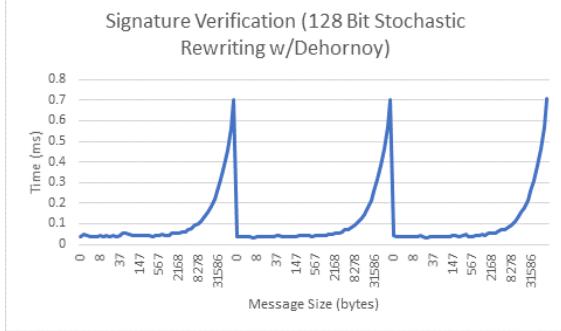


Fig. 11: SUPERCOP output for verifying 128-bit signatures with Stochastic Rewriting and Dehornoy

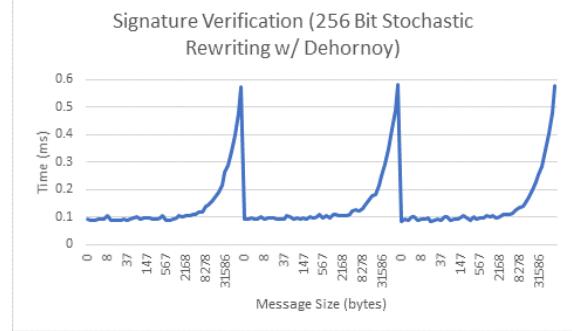


Fig. 12: SUPERCOP output for verifying 256-bit signatures with Stochastic Rewriting and Dehornoy

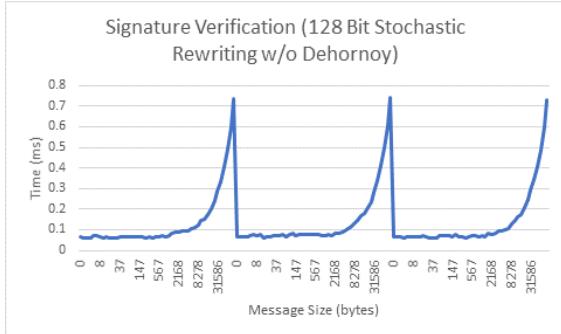


Fig. 13: SUPERCOP output for verifying 128-bit signatures with Stochastic Rewriting, without Dehornoy

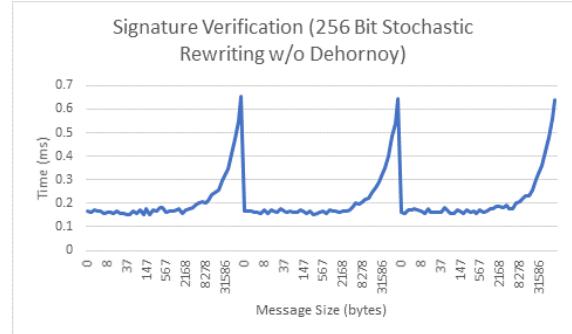


Fig. 14: SUPERCOP output for verifying 256-bit signatures with Stochastic Rewriting, without Dehornoy

Message Size (B)	567	709	887	1109	1387	1734	2168	2711	3389	4237	5297	6622
Cycles	116928	124848	111208	122656	123960	132096	135232	140480	155616	172328	201080	214144
Message Size (B)	8278	10348	12936	16171	20214	25268	31586	39483	49354	61693	77117	96397
Cycles	244896	290880	332376	392016	475392	567856	679272	828904	1010096	1235120	1519880	1881312

Table 3: SUPERCOP cycle counts for signature verification

The same pattern, where message size affects verification speed, can also be seen for 256-bit signatures in Figure 10. Similarly, Figures 11, 12, 13, and 14 all exhibit the same structure.

The main difference between them all is the baseline computation time. The graphs show that Dehornoy vs non-Dehornoy is the main component to speed (which is to be expected, considering validation computation time is linear in the length of the signature, and Dehornoy reduces the signature size). Still, without Dehornoy the base verification time is just under 0.1ms for 128-bit signatures and about 0.17ms for 256-bit signatures, and when Dehornoy is applied those times are reduced to 0.05ms and 0.1ms.

In addition to working on the target platform with SUPERCOP, we also took an average-length signature at 128-bit security level and ran the verification computation on various embedded processors used on devices associated with the Internet of Things. See Table 4 for the raw data.

Platform	Bits	Clock (MHz)	ROM	RAM	Cycles	Time (ms)
8051	8	24.5	3370	312	864101	35.3
MSP430	16	8	3244	236	370944	46
ARM Cortex M3	32	48	2952	272	275563	5.7
FPGA		50	1720(ALM)		2500	0.05

Table 4: Raw WalnutDSA performance data for verifying 128-bit signatures

C Expected Security Strength

At this time the best-known attack against WalnutDSA is a brute force search. See [1] for a full security analysis. The summary is that for a given security level (SL):

1. A public key is a pair of $N \times N$ matrices of elements in F_q , which implies there are a maximum of q^{N^2} potential options per matrix. However, due to the construction, this is reduced to a minimum of $q^{N(N-3)}$ potential matrices. We have chosen N and q such that this value exceeds 2^{SL} possible public keys.
2. A secret key is a pair of braidwords in B_N of length ℓ . The number ℓ is chosen such that there are at least 2^{SL} unique braids when randomly creating a braid of length ℓ (prior to free reduction).
3. Cloaking elements are chosen in words of length L of the pure braid generators. The value of L is chosen such that there are at least 2^{SL} possible words.
4. Reversing the rewriting schemes is also a brute-force problem which far exceeds 2^{SL} operations to reverse.

We have increased our values for L and ℓ by 25% beyond the minimum required to meet the desired security level, both for 128- and 256-bit (conventional) security. This increase is purely for future proofing against minor errors or miscalculations in the number of possible braid or braidwords or improvements in enumeration techniques.

We believe that WalnutDSA is subject to Grover so we expect that the quantum security is half of the conventional security.

D Advantages and Limitations

The main advantages of WalnutDSA are that key generation and signature validation are extremely fast, even on small, constrained devices. These functions can be implemented in very little code and compile down to very small targets. Indeed, the raw signature validation on a 48MHz ARM Cortex M3 can complete in 5.7ms in compiled C software. Due to its nature, signature validation can easily be computed even on 16- or 8-bit processors with limited RAM and ROM and decent performance.

The main limitation of WalnutDSA is that signature generation is more expensive because the known braid rewriting techniques are more computationally intensive.

E Known Attacks

If $\text{Priv}(S) = \text{Priv}(S')$ then there is a factoring attack that can potentially create a valid signature by combining multiple signatures to create new words. However, the signatures generated by this attack are orders of magnitude longer than a valid WalnutDSA signature (estimated at a length 2^{32} or longer), and this attack is completely defeated by ensuring that $\text{Priv}(S) \neq \text{Priv}(S')$. See [1] for an analysis of this attack.

The next best-known attack against WalnutDSA is a brute force search. See [1] for a full security analysis, including a proof of security under EUF-CMA.

There are no known other attacks against WalnutDSA as of this writing.