NIST Submission: Xifrat1-Sign.I DSS

```
Name of the Algorithm:
Xifrat1-Sign.I

Submitters:

Jianfang "Danny" Niu

Email:
  dannyniu {at} hotmail {dot} com
Phone:
  +86 173 1763 5231
Organization:
  This work is done without the endorsement of my employer
Room Address:
  Tech Department, 3F, Number 479 Shangzhong Road,
City Address:
  Xuhui District, Shanghai, China (Mainland).


Daniel Enrique Náger Piazuelo

Email:
  daniel {dot} nager {at} gmail {dot} com
```

!NOTE! The HTML rendering of this document is not authoritative, and readers seeking a stable reference should look for the PDF version published at official sources.

Table of Contents

Figures

1. Algorithm Specification

The Xifrat1 family of cryptosystems are based on a randomly generated abelian quasigroup of 16 elements. From which, 3 layers of increasingly large abelian quasigroups are constructed such that recovering the input operand and computing the output without full knowledge of either operand become increasingly difficult at exponential rate.

The Xifrat1-Sign.I digital signature scheme instance is one member of the Xifrat1 family of cryptosystems. It may be of separate interest should NIST elect to standardize the KEM/PKE Xifrat1-Kex.I, as it's more compact than Kyber in terms of communication bandwidth.

1.1. The quasigroup and building block functions

In this section, we present the quasigroup table, discuss the property of restricted commutativity and its generalization (which we will be using), and present a construction that enlarges the quasigroup.

1.1.1. The restricted commutative quasigroup

The quasigroup we're considering has the following properties:

- Non-Associative *In General*: that is, for most cases, $(ab)c \neq a(bc)$
- Non-Commutative *In General*: that is, for most cases, $ab \neq ba$
- Restricted-Commutativity: that is, for all cases, $(ab)(cd) = (ac)(bd)$

Additionally, some properties are needed for basic security:

- The quasigroup table should overall be not symmetric;
- The quasigroup table should not have any fixed points;

We observed that, in Xifrat0, as well as the StackExchange post that sparked all these discussion, the quasigroup tables had a regularity that, for each diagonal pair of equal table cells, the opposite diagonal is also equal. This appears to be a necessary but not sufficient condition for a power-of-2 table to be restricted-commutative; as for non-power-of-2 tables, experiment had shown this property does not apply to them.

We used diagonal property for optimization and created a new program that searched for a random quasigroup table with the seed "`xifrat - public-key cryptosystem`" which is the same one that's used in Xifrat0. Additionally in July 2022, we applied an aggressive assignment optimization to the QGGen-v2 program, and was able to obtain a 16-by-16 quasigroup in about 10 to 15 minutes on an Intel Core i3-8100B (3.6GHz) process. The program is single-threaded to ensure the result is deterministic.

The source code for the new program can be found at our online git repository: https://github.com/dannyniu/xifrat The new table is as follow:

```
// Quasigroup generated using the new program //
  10  11   0   3  12   4   1   5  15   6   8  14   2   9   7  13
  15   8   9   7   2  13   5   1  10  14  11   6  12   0   3   4
   2   3   6  11  15   5  13   4  12   0   7   9  10  14   8   1
   0   5  10   4  14   3   8  11   9   2   1  12   6  15  13   7
   8  15   1  12   3  14   0   9  11  13  10   4   7   5   2   6
   6   4   2   5   9  11   7   3  14  10  13  15   0  12   1   8
  13  14   7   9   5  15   2  12   4   8   6  11   1   3   0  10
  12   7  14   8  10   1   4  13   2   9   3   0  15   6  11   5
   5   0  11   6  13   2  15  10   1   3   9   7   4   8  14  12
  14  13  12   1   0   8   3   7   6  15   4  10   9   2   5  11
   1   9   8  14   4  12  10  15   5   7   0   3  13  11   6   2
   7  12  13  15  11   9   6  14   3   1   2   5   8   4  10   0
   9   1  15  13   6   7  11   8   0  12   5   2  14  10   4   3
   4   6   3   0   1  10  12   2  13  11  14   8   5   7   9  15
  11  10   5   2   7   6   9   0   8   4  15  13   3   1  12  14
   3   2   4  10   8   0  14   6   7   5  12   1  11  13  15   9
```

The operation ab evaluates to the table cell at a'th row and b'th column, in 0-based index.

1.1.2. The generalized restricted commutativity

Now we introduce an important property, that is both useful, and comes naturally from restricted-commutativity: the generalized restricted commutativity.

Theorem 1. *Left-associativity of distributiveness*

That is:
$(a_1\, b_1)(a_2\, b_2)\ ...\ (a_n\, b_n) = (a_1\, a_2\ ...\ a_n)(b_1\, b_2\ ...\ b_n)$

Proof:

Observe a case of 3 pairs: $(ab)(cd)(ef)$.
due to restricted commutativity: $(ac)(bd)(ef)$,
next, substitute $g=(ac)$, $h=(bd)$, we have:
$(gh)(ef)$,
again, due to restricted commutativity, we have: $(ge)(hf)$,
substitute back, we have $(ace)(bdf)$,
generalizing recursively, we have **Theorem 1.**

Property 1. *Generalized Restricted-Commutativity*

That is:
$(x_{1,1}\ x_{1,2} \cdots x_{1,n})\ (x_{2,1}\ x_{2,2} \cdots x_{2,n}) \cdots (x_{m,1}\ x_{m,2} \cdots x_{m,n}) =$
$(x_{1,1}\ x_{2,1} \cdots x_{m,1})\ (x_{1,2}\ x_{2,2} \cdots x_{m,2}) \cdots (x_{1,n}\ x_{2,n} \cdots x_{m,n})$

Proof: From **Theorem 1.**, we have

$(x_{1,1}\ x_{1,2} \cdots x_{1,n})\ (x_{2,1}\ x_{2,2} \cdots x_{2,n}) \cdots (x_{m,1}\ x_{m,2} \cdots x_{m,n}) =$
$((x_{1,1}\ x_{2,1})(x_{1,2}\ x_{2,2}) \cdots (x_{1,n}\ x_{2,n}))\ (x_{3,1}\ x_{3,2} \cdots x_{3,n}) \cdots (x_{m,1}\ x_{m,2} \cdots x_{m,n}) =$
$(\ ((x_{1,1}\ x_{2,1})\ x_{3,1})\ ((x_{1,2}\ x_{2,2})\ x_{3,2}) \cdots (x_{1,n}\ x_{2,n} \cdots x_{m,n})) =$
$(x_{1,1}\ x_{2,1} \cdots x_{m,1})\ (x_{1,2}\ x_{2,2} \cdots x_{m,2}) \cdots (x_{1,n}\ x_{2,n} \cdots x_{m,n})$

1.1.3. The Blk block function

The Blk block function is defined to enlarge the quasigroup - it operates on vector of 16 quartets bitstrings. This is 64-bit in total, which we fit in least-significant- bit&byte -first order.

Figure 1.1. The algorithm for the Blk function

- Input: $A=(a_0\ a_1 \cdots a_{15})$, $B=(b_0\ b_1 \cdots b_{15})$
- Output: $C=(c_0\ c_1 \cdots c_{15})$

Steps:

- $c_0 = (a_0\ a_1 \cdots a_{15})\ (b_0\ b_1 \cdots b_{15})\ (a_0\ a_1 \cdots a_{15})\ (b_0\ b_1 \cdots b_{15})$
- $c_1 = (a_1\ a_2 \cdots a_0)\ (b_1\ b_2 \cdots b_0)\ (a_1\ a_2 \cdots a_0)\ (b_1\ b_2 \cdots b_0)$
- $c_2 = (a_2\ a_3 \cdots a_1)\ (b_2\ b_3 \cdots b_1)\ (a_2\ a_3 \cdots a_1)\ (b_2\ b_3 \cdots b_1)$
- ...
- $c_{15} = (a_{15}\ a_0 \cdots a_{14})\ (b_{15}\ b_0 \cdots b_{14})\ (a_{15}\ a_0 \cdots a_{14})\ (b_{15}\ b_0 \cdots b_{14})$

Programmatically, A, B, and C are represented as the `uint64_t` data type.

1.1.4. The Vec, Red, and Dup functions

The purpose of the Vec function is the same as that of the Blk function, except it works over a larger domain. The Vec function takes 2 vectors of 6 64-bit slices. each are 384-bit long, and return 1 vector as result. The construction of Vec is structurally similar to Blk.

Within the Vec function, each of the 64-bit slices are "hashed" in the Blk function, and applied sequentially twice interlaced with the other operand. An obvious flaw is that, if we can *individually* brutal-force the slices, then we can evaluate either operand without knowing it in full, which leads to a fatal break.

This is why, another layer is needed, which we call Dup. The operands for the Dup function are in the form of bi-gram of vectors, where the vectors are operands to the Vec function. The purpose of Dup is, yet again, the same as Blk as well as Vec, but this time, the 6 slices are "hashed", requiring attacker to brutal force $6 \times 64 = 384$ bits. While this may be an overkill for some scenario, we leave this as an overhead in case any powerful cryptanalytic attack is discovered.

To show how parameters scale, we specify a variant of the Vec function - the Red function (meaning 'Reduced'), which operates on vectors of 4 64-bit slices. This function is also meant to be used under Dup. We've implemented this variant in the "ReduceSec_Implementation" purely for investigative purposes such as benchmarking and cryptanalysis, and we do not intend this variant be standardized.

Figure 1.2. The algorithm for the Vec function

- Input: A=(A_0 A_1 ... A_5) , B=(B_0 B_1 ... B_5)
- Output: C=(C_0 C_1 ... C_5)

Steps:

- $C_0 = (A_0\, A_1\, ...\, A_5)\ (B_0\, B_1\, ...\, B_5)\ (A_0\, A_1\, ...\, A_5)\ (B_0\, B_1\, ...\, B_5)$
- $C_1 = (A_1\, A_2\, ...\, A_0)\ (B_1\, B_2\, ...\, B_0)\ (A_1\, A_2\, ...\, A_0)\ (B_1\, B_2\, ...\, B_0)$
- $C_2 = (A_2\, A_3\, ...\, A_1)\ (B_2\, B_3\, ...\, B_1)\ (A_2\, A_3\, ...\, A_1)\ (B_2\, B_3\, ...\, B_1)$
- ...
- $C_5 = (A_5\, A_0\, ...\, A_4)\ (B_5\, B_0\, ...\, B_4)\ (A_5\, A_0\, ...\, A_4)\ (B_5\, B_0\, ...\, B_4)$

Figure 1.3. The algorithm for the Red function

- Input: A=(A_0 A_1 ... A_3) , B=(B_0 B_1 ... B_3)
- Output: C=(C_0 C_1 ... C_3)

Steps:

- $C_0 = (A_0\, A_1\, A_2\, A_3)\ (B_0\, B_1\, B_2\, B_3)\ (A_0\, A_1\, A_2\, A_3)\ (B_0\, B_1\, B_2\, B_3)$
- $C_1 = (A_1\, A_2\, A_3\, A_0)\ (B_1\, B_2\, B_3\, B_0)\ (A_1\, A_2\, A_3\, A_0)\ (B_1\, B_2\, B_3\, B_0)$
- $C_2 = (A_2\, A_3\, A_0\, A_1)\ (B_2\, B_3\, B_0\, B_1)\ (A_2\, A_3\, A_0\, A_1)\ (B_2\, B_3\, B_0\, B_1)$
- $C_3 = (A_3\, A_0\, A_1\, A_2)\ (B_3\, B_0\, B_1\, B_2)\ (A_3\, A_0\, A_1\, A_2)\ (B_3\, B_0\, B_1\, B_2)$

Figure 1.4. The algorithm for the Dup function

- Input: $A=(A_0\ A_1)$, $B=(B_0\ B_1)$
- Output: $C=(C_0\ C_1)$

Steps:

- $C_0 = (A_0\ A_1)\ (B_0\ B_1)\ (A_0\ A_1)\ (B_0\ B_1)$
- $C_1 = (A_1\ A_0)\ (B_1\ B_0)\ (A_1\ A_0)\ (B_1\ B_0)$

Programmatically, the operands to Vec and Dup functions are represented as array types: `uint64_t[6]` and `uint64_t[12]` . For the Dup function, slice indicies 0~5 corresponds to the vector at index 0 of the bi-gram and indicies 6~12 corresponds to that at 1. We will call operands to the Dup function "cryptograms" of the Xifrat schemes.

For ease of readability, we denote the Dup function as $D(a,b)$ and $D(D(a,b),c)$ as $(a \bullet b \bullet c)$.

1.2. The Xifrat1-Sign.I Digital Signature Scheme

In this section, we present the Xifrat1-Sign.I digital signature scheme. We use a hash function, which is instantiated with the XOF SHAKE-256. We take its initial 768-bit output, interpret it as 12 64-bit unsigned integers in little-endian. We denote this hash function as $\mathrm{Hx}_{768}(m)$.

Figure 1.5. Xifrat1-Sign.I Key Generation

1. Uniformly randomly generate 3 cryptograms: c, k, and q ,
2. Compute $p_1 = D(c,k)$, $p_2 = D(k,q)$,
3. Return public-key $pk = (\ c\ , p_1\ , p_2\ )$ and private-key $sk = (\ c\ , k\ , q\ )$.

Figure 1.6. Xifrat1-Sign.I Signature Generation

1. **Input:** m - the message
2. Compute $h = \mathrm{Hx}_{768}(m)$,
3. Compute $s = D(h,q)$,
4. Return s ,

Figure 1.7. Xifrat1-Sign.I Signature Verification

```
1. Input: $m$ - the message , $s$ - the signature
2. Compute $h = \mathrm{Hx}_{768}(m)$ ,
3. Compute $t_1 = D( p_1 , s )$ ,
4. Compute $t_2 = D( D(c,h) , p_2 )$ ,
5. If $t_1 = t_2$ return [VALID] ; otherwise return [INVALID].
```

The proof of correctness of the scheme is as follow:

$t_1 = D(p_1 , s) = D(D(c,k) , D(h,q))$
$t_2 = D(D(c,h) , p_2) = D(D(c,h) , D(k,q))$
By restricted commutativity, we have $t_1 = t_2$.

Parameters	**Xifrat1-Sign.I**	**Reduced-Security Variant**
private key bytes	480	320
public key bytes	288	192
signature bytes	96	64

2. Performance and Efficiency

The dominant part of Xifrat computation is the computation of the Dup function. We compiled the codes constituting the Xifrat abelian quasigroup computation with the `-O` optimization flag, and ran it for 256x6 times, here's a rough benchmarking result. For Apple M1, we do not have public documentation that describes its clock speed. For the Intel CPU benchmark, we carried it out on Windows Subsystem for Linux - WSLv2, using a Debian distribution. In both cases, we used the Clang/LLVM compiler.

Dup instantiated with Vec

CPU model	**Cycles per 256x6 iterations of `Dup`**	**Cycles per once `Dup`**	**Times per 256x6 iterations**	**Times per once**
Apple M1	27727871	18052.00	27.73s	18.05ms
Intel Core i7-10700F	31826300	20720.25	31.83s	20.72ms

For key generation, ignoring the cost of invoking PRNG, it takes 2 `Dup` computations, therefore 1 keygen takes about 30ms~40ms. For signing operation, ignoring the cost of hashing, only 1 `Dup` computation is required. Signature verification however takes 3 `Dup` computation, which is the most expensive of these, and it takes about 45ms~60ms. While these costs are high, we believe that if Xifrat turn out to be secure, the merit of its compactness will outweight performance drawbacks.

We performed similar benchmark with Dup instantiated with Red, and the results are as follow:

Dup instantiated with Red

CPU model	**Cycles per 256x6 iterations of `Dup`**	**Cycles per once `Dup`**	**Times per 256x6 iterations**	**Times per once**
Apple M1	12057375	7849.85	12.06s	7.852ms
Intel Core i7-10700F	14480765	9427.58	14.48s	9.427ms

The working context for Xifrat1-Sign.I only need to contain countably finite number of cryptograms, additional cryptograms on the stacks of internal subroutines are also few. This is a drastic contrast when compared to schemes that derive large amount of data from a "seed value" or that derive some kind of steep tree structure.

3. Security and Known Attacks

The Xifrat1-Sign.I instance is expected to have category III security, which corresponds to brutal-forcing a 192-bit symmetric-key encryption algorithm on classical computers. Xifrat1-Sign.I uses a XOF with 256-bit security, and takes 768-bit output, of which, two 384-bit halves are processed by the Dup function. The Dup function is expected to have 384-bit classical security against brutal-force attack, against input operand recovery and evaluation without full knowledge of either operand. We've doubled it from 192-bit for 2 important reasons: 1.) to prevent partial forgery against the hash digest, and 2.) to deter potentially unknown future mathematic and group-theoretic cryptanalysis that may be discovered for both classical and quantum computer; thirdly, the call for proposal from NIST had requested submitters to be conservative about evaluating security, which we count as an extra reason.

It is the opinion of the author, that group-theoretic cryptography is promising, but is still in its infancy, with mathematical structures being proposed all the time without anything remarkable turnning up. Many previous attempts had failed: Algebraic Eraser[BzBT16], WulnutDSA[HKM+18], and even SIKE[CD22].

3.1. Attack 1: Evaluate without Full Knowledge of 1 Operand

The design of the mixing functions at each layer has 2 phases - the cycling phase where 2 intermediate values u and v are computed from vector elements of a and b; the alternating phase of $uvuv$. By generalized restricted-commutativity, the cycling and alternating phases can be computed in either order and results in the same value output. The resulting template mixing function preserves the restricted-commutativity property from a lower layer to a higher layer

The first attack we discuss is the "evaluation without full knowledge of either operand" attack. This attack was present in a most fatal form in Xifrat0, and was quickly broken [Niu21] back then.

The same apply to Xifrat1. Recall that the "Blk" function works over a vector of 16 quartets. If we can find either of the intermediate u or v , then we can use that knowledge to compute that function - because the alternating phase works parallelly over the vector of quartets. This attack has to be blocked at the cycling phase of a higher layer.

The cycling phase at a higher layer mixes together the vector elements of the lower layer, making it necessary to recover the vector in its entirety to be able to compute the mixing function. This is why at 384 bits, we still need a "Dup" layer on top of the "Vec" layer.

3.2. Attack 2: Group Theoretic Analysis

This section discusses group theoretic cryptanalysis on Xifrat1.

As we said in [Niu22], the 16x16 latin square was chosen randomly; and we assume its quasigroup operation is also random, in the sense that it *behave as if randomly*.

It's a fact [Bruck44] [Murdoch39] [Toyoda41] that, any quasigroup, like that we've been using, can be decomposed into a group with 2 automorphisms:

$f(a,b) = g(a) + h(b) + c$ where f is the quasigroup operation, g and h are 2 automorphisms which we assume *are independent of each other* and *behave as if randomly*, and c is a constant from the quasigroup set, and $+$ is the operation of the decomposed group.

Now let's see an example:

$g(x) + h(y) = u$

$g(y) + h(x) = v$

If g and h are truely random, then the only way to find x , y from u , v would be to try every possible solution and verify each of them to find out. Because we generated our 16x16 quasigroup randomly, we assume that the underlaying automorphisms fulfills this property. It is further assumed, that composition of randomly-behaving maps are also randomly-behaving.

The Blk function can now be reverted, by first searching 16 independent quartets from the alternating phase, then solving the "cycling" equations system, which consists of 16 group equations, each with 16 automorphisms that we had assumed to be "randomly-behaving".

There are 2 crucial assumptions we make, as the basis for believing that the mixing funcitons at higher layer are more difficult to to invert than the Blk function.

1. The group automorphisms exploitation is the most efficient attack applicable to the cycling-alternating mixing function formula *when assuming* that there is a efficient way to evaluate the automorphisms.
2. There is no efficient way to find or evaluate larger group automorphisms from the group and automorphisms underlaying the smaller abelian quasigroup *assuming* the underlaying abelian quasigroup is randomly-behaving.

Additionally, we assume that the expansion result of abelian quasigroup formula at higher layer into group automorphisms is no more efficiently solvable than applying layer-by-layer approach according to the preceding list, as the expansion of the formula terms is polynomial (which we believe makes the solution of the equasions system super-exponential, but we have yet no way of being sure).

4. Advantages and Limitations

The biggest advantage of Xifrat1-Sign.I (and in general, any scheme in the Xifrat1 family), is compactness. The signature size of the DSS is 96 bytes. The size of the public and private key sizes are on par with that of RSA keypairs encoded using the PKCS#1 ASN.1 module. Another advantage is that, the key generation of Xifrat1-Sign.I is constant-time, and unlike Falcon or NTRU, Xifrat1-Sign.I doesn't require computing any "inverse" element. Finally, it is a guess of the submitters that due to the arithmetic property of the Dup function, it may be possible to implement "implicit certificate" with Xifrat1-Sign.I.

One disadvantage is the signature verification time is long, although not too long. Another major disadvantage is that, the underlaying primitive and security assumption are arcane and poorly understood.

Annex A. References

- [BzBT16] Adi Ben-Zvi, Simon R. Blackburn, Boaz Tsaban; *A Practical Cryptanalysis of the Algebraic Eraser*; In: Robshaw, M., Katz, J. (eds) Advances in Cryptology – CRYPTO 2016. CRYPTO 2016. Lecture Notes in Computer Science(), vol 9814. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-53018-4_7
- [HKM+18] Daniel Hart, DoHoon Kim, Giacomo Micheli, Guillermo Pascual-Perez, Christophe Petit, Yuxuan Quek; *A Practical Cryptanalysis of WalnutDSA^{TM}*; In: Abdalla, M., Dahab, R. (eds) Public-Key Cryptography – PKC 2018. PKC 2018. Lecture Notes in Computer Science(), vol 10769. Springer, Cham. https://doi.org/10.1007/978-3-319-76578-5_13
- [CD22] Wouter Castryck, Thomas Decru; 2022-07 *An efficient key recovery attack on SIDH* https://eprint.iacr.org/2022/975

- [Bruck44] Richard H. Bruck; *Some Results in the Theory of Quasigroups*; In: *Transactions of the American Mathematical Society* 55.1 (1944), pp. 19-52.
- [Murdoch39] David C. Murdoch; *Quasi-Groups Which Satisfy Certain Generalized Associative Laws*; In: *American Journal of Mathematics* 61.2 (1939), pp.509-522.
- [Toyoda41] Koshichi Toyoda; *On axioms of linear functions*; In: *Proceedings of the Imperial Academy* 17.7 (1941), pp.221-227.
- [NN21] Daniel Nager, and Jianfang "Danny" Niu; 2021-04 *Xifrat - Compact Public-Key Cryptosystems based on Quasigroups*; https://ia.cr/2021/444
- [Niu21] Jianfang "Danny" Niu; 2021-04 *Xifrat Cryptanalysis - Compute the Mixing Function Without the Key*; https://ia.cr/2021/487
- [Niu22] Jianfang "Danny" Niu; 2022-04 *Resurrecting Xifrat - Compact Cryptosystems 2nd Attempt*; https://ia.cr/2022/429