Resurrecting Xifrat - Compact Cryptosystems 2nd Attempt

Xifrat was a group-theoretic public-key cryptosystem based on a quasigroup with the special property of "restricted-commutativity". It was broken within half a month of its publication, due to a mistake made in the "mixing" function. In this paper, we revisit the design decisions made, proposing new constructions, and attempt (again) to build secure digital signature schemes and key encapsulation mechanisms. If the schemes can be proven secure, then this will be the most compact and the most efficient post-quantum cryptosystem ever proposed to date.

Authors: Jianfang "Danny" Niu (dannyniu {at} hotmail {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

1. Background **3**

2. The quasigroup and building block functions **3**
2.1. The restricted commutative quasigroup 4
2.2. The generalized restricted commutativity 4
2.3. The Blk block function 5
2.4. The Vec and Dup functions 6

3. The digital signature and key exchange schemes **7**
3.1. The Xifrat1-Sign.I DSS 7
3.2. The Xifrat1-Kex.I KEM 9

Annex A. References **10**

Figures

Figure 2.1. The algorithm for the Blk function 6
Figure 2.2. The algorithm for the Vec function 7
Figure 2.3. The algorithm for the Dup function 7
Figure 3.1. Xifrat1-Sign.I Key Generation 8
Figure 3.2. Xifrat1-Sign.I Signature Generation 8
Figure 3.3. Xifrat1-Sign.I Signature Verification 8
Figure 3.4. Xifrat1-Kex.I Key Generation 9
Figure 3.5. Xifrat1-Kex.I Encapsulation 10
Figure 3.6. Xifrat1-Kex.I Decapsulation 10

1. Background

In this paper, we propose 2 group-theoretic compact public-key cryptosystems - a key ecapsulation mechanism (KEM) and a digital signature scheme (DSS). These cryptosystems are aimed at achieving at least 192-bit post-quantum security.

Previously, most practical compact cryptosystems are based on elliptic curves, whether it's the pre-quantum ones based on EC discrete logarithm [SEC#1], or elliptic curve isogenies. The former are vulnerable to Shor's algorithm on quantum computers; while the latter are suffering from some performance problems. For example, SIKE [SIKE] is a NIST 3rd-round KEM/PKE alternate candidate that executes in time that's an order of magnitude longer than lattice-based ones such as Kyber [Kyber] and Saber [Saber] , although, such difference is not too noticable; likewise, a later design that didn't manage to get into the NIST PQC project - SQISign [SQISign] from Oct 2020 has signing time that's over 2 seconds long on modern workstation PCs.

Group-theoretic cryptography are, in the opinion of the author, still in its infancy - with closures of various theoretical structures and properties being proposed and analyzed without anything remarkable turning up. Algebraic Eraser [AE] being a prominent example based on braid group for key agreement that had failed to be standardized in an ISO/IEC standard; WalnutDSA [WalnutDSA] being another prominent example that didn't pass the 1st round in the NIST PQC project. Both due to security issues.

Xifrat aims to provide PQC schemes that're compact through use of a class of groupoid with the property of *restricted-commutativity*. Such groupoid was previously proposed in [NN21], however, a critical error was made in designing the "mixing" function, which resulted in a total break, just half a month after its publication. We retroactively name the scheme in that paper Xifrat0 (and Xifrat0-Kex and Xifrat0-Sign). In this paper, we revisit the design decisions, and devise new constructions that can be used securely (or more accurately: *may* be used securely **if** the underlying primitive can be proven secure).

2. 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 in the PKC schemes), and present a construction that enlarges the quasigroup.

2.1. 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. Although we had hoped for that our optimization can make it possible to find a 16-by-16 quasigroup, the poly-exponential-time complexity ultimately convinced us to give up.

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 //
  2   0   4   3   5   7   1   6
  1   5   3   4   0   6   2   7
  7   4   0   5   3   2   6   1
  0   2   7   6   1   4   5   3
  3   6   1   2   7   5   4   0
  6   3   5   0   4   1   7   2
  4   7   2   1   6   0   3   5
  5   1   6   7   2   3   0   4
```

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

We propose the **1st open problem** of this paper: Can we find a verifiably random 16-by-16 quasigroup table? Can we find one efficiently?

2.2. 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:
$(x_{1,1}\ x_{1,2}\ ...\ x_{1,n})\ (x_{2,1}\ x_{2,2}\ ...\ x_{2,n})\ ...\ (x_{m,1}\ x_{m,2}\ ...\ x_{m,n}) =$
$(x_{1,1}\ x_{2,1}\ ...\ x_{m,1})\ (x_{1,2}\ x_{2,2}\ ...\ x_{m,2})\ ...\ (x_{1,n}\ x_{2,n}\ ...\ x_{m,n})$

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

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

2.3. The Blk block function

The Blk block function is defined to enlarge the quasigroup - it operates on vector of 21 tritet bitstrings. This is 63-bit in total, which we fit in least-significant- bit&byte -first order.

Figure 2.1. The algorithm for the Blk function

- Input: $A=(a_0\ a_1\ ...\ a_{20})$, $B=(b_0\ b_1\ ...\ b_{20})$
- Output: $C=(c_0\ c_1\ ...\ c_{20})$

Steps:

- $c_0 = (a_0\ a_1\ ...\ a_{20})\ (b_0\ b_1\ ...\ b_{20})\ (a_0\ a_1\ ...\ a_{20})\ (b_0\ b_1\ ...\ b_{20})$
- $c_1 = (a_1\ a_2\ ...\ a_0)\ (b_1\ b_2\ ...\ b_0)\ (a_1\ a_2\ ...\ a_0)\ (b_1\ b_2\ ...\ b_0)$
- $c_2 = (a_2\ a_3\ ...\ a_1)\ (b_2\ b_3\ ...\ b_1)\ (a_2\ a_3\ ...\ a_1)\ (b_2\ b_3\ ...\ b_1)$
- ...
- $c_{20} = (a_{20}\ a_0\ ...\ a_{19})\ (b_{20}\ b_0\ ...\ b_{19})\ (a_{20}\ a_0\ ...\ a_{19})\ (b_{20}\ b_0\ ...\ b_{19})$

Programmatically, A, B, and C are represented as the `uint64_t` data type, with the top bit clear.

There is one problem with the Blk function, that is, when it's given 2 vectors of repeated tritets, it produces a vector of repeated tritet. This is due to such input would produce same array of operations at every lane of output tritet. At the moment, we do not know if this is exploitable in the actual KEM and DSS scheme.

2.4. The Vec 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 7 63-bit slices. each are 448-bit long with 441 effective bits, and return 1 vector as result. The construction of Vec is structurally similar to Blk.

Within the Vec function, each of the 63-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 had been an oversight in the previous versions of this paper, which we fix now, by appending the Roman numeral ".I" to the name of both schemes.)

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 7 slices are "hashed", requiring attacker to brutal force $7 \times 63 = 441$ bits. While this is a overkill for almost every scenario, we leave this as an overhead in case any powerful cryptanalytic attack is discovered.

Figure 2.2. The algorithm for the Vec function

- Input: $A=(A_0\,A_1\,...\,A_6)$, $B=(B_0\,B_1\,...\,B_6)$
- Output: $C=(C_0\,C_1\,...\,C_6)$

Steps:

- $C_0 = (A_0\,A_1\,...\,A_6)\,(B_0\,B_1\,...\,B_6)\,(A_0\,A_1\,...\,A_6)\,(B_0\,B_1\,...\,B_6)$
- $C_1 = (A_1\,A_2\,...\,A_0)\,(B_1\,B_2\,...\,B_0)\,(A_1\,A_2\,...\,A_0)\,(B_1\,B_2\,...\,B_0)$
- $C_2 = (A_2\,A_3\,...\,A_1)\,(B_2\,B_3\,...\,B_1)\,(A_2\,A_3\,...\,A_1)\,(B_2\,B_3\,...\,B_1)$
- ...
- $C_6 = (A_6\,A_0\,...\,A_5)\,(B_6\,B_0\,...\,B_5)\,(A_6\,A_0\,...\,A_5)\,(B_6\,B_0\,...\,B_5)$

Figure 2.3. 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[7]` and `uint64_t[14]` . For the Dup function, slice indicies 0~6 corresponds to the vector at index 0 of the bi-gram and indicies 7~13 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 $(a \cdot b \cdot c)$.

3. The digital signature and key exchange schemes

3.1. The Xifrat1-Sign.I DSS

In this section, we present the Xifrat1-Sign.I digital signature scheme. The general structure is similar to Xifrat0-Sign, but uses the Dup function to actually achieve unforgeability.

As with Xifrat0-Sign, we use a hash function, which is instantiated with the XOF SHAKE-256. We take its initial 896-bit output, interpret it as 14 64-bit unsigned integers in little-endian, and clear each of their top bits. We denote this hash function as $\mathrm{Hx}_{896\text{-}14}(m)$.

Figure 3.1. 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 3.2. Xifrat1-Sign.I Signature Generation

1. **Input:** m - the message
2. Compute $h = \mathrm{Hx}_{896\text{-}14}(m)$,
3. Compute $s = D(h,q)$,
4. Return s ,

Figure 3.3. Xifrat1-Sign.I Signature Verification

1. **Input:** m - the message , S - the signature
2. Compute $h = \mathrm{Hx}_{896\text{-}14}(m)$,
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	
private key bytes	560
public key bytes	336
signature bytes	112

3.2. The Xifrat1-Kex.I KEM

In this section, we present construction for key exchange.

As we've had the generalized restricted-commutativity property, we can construct a 9-variable key agreement scheme by laying out them in a square like this:

$$(abc)(def)(ghi) = (adg)(beh)(cfi)$$

For ease of visualized intuiation, we lay them graphically:

```
// 9 variables //
a b c
d e f
g h i
```

The sum is identical regardless whether the matrix is evaluated row-first or column-first. Thus we take the middle column as the public key and the middle row as the ciphertext; b, h as "server-side" private key, and d, f as "client-side" secret share; The variables a, c, e, g, i are used as public information. The public key and the ciphertext are both in some kind of "sandwich" structure we mentioned in the previous section, which we believe makes recovering private keys impossible.

It is obvious at this point that the public information can be derived from a seed using some extendable output function (XOF), (prior art: [NewHope]). We instantiate such XOF with SHAKE-128. We take 896-bit in turn, interpret it as 14 64-bit unsigned integers in little-endian and clear each of their top bits, and generate 5 of these and assign them to a, c, e, g, i in order. We denote this XOF as $\mathrm{Hx}_{[896\text{-}14]\times 5}(seed)$.

Figure 3.4. Xifrat1-Kex.I Key Generation

1. Uniformly randomly generate choose a 8-octet $seed$,
2. Generate a, c, e, g, i using $\mathrm{Hx}_{[896]\times 5}(seed)$,
3. Uniformly randomly generate 2 cryptograms b, h ,
4. Compute $p = (b \cdot e \cdot h)$,
5. Return $pk = (\ seed\ ,\ p\ )$ as the public key and $sk = (\ seed\ ,\ b\ ,\ h\ )$ as the private key.

Figure 3.5. Xifrat1-Kex.I Encapsulation

1. Expand $seed$ into a, c, e, g, i using $\mathrm{Hx}_{[896]\times 5}(seed)$,
2. Uniformly randomly generate 2 cryptograms d, f ,
3. Compute $ss = (a \cdot d \cdot g) \cdot pk \cdot (c \cdot f \cdot i)$,
4. Compute $ct = (d \cdot e \cdot f)$,
5. Return ss as shared secret and ct as ciphertext .

Figure 3.6. Xifrat1-Kex.I Decapsulation

1. Return $ss = (a \cdot b \cdot c) \cdot ct \cdot (g \cdot h \cdot i)$ as shared secret.

Parameters	
private key bytes	232
public key bytes	120
ciphertext bytes	112

Annex A. References

- [AE] Derek Atkins, P. Gunnells; 2015; *Algebraic Eraser (TM) : A lightweight, efficient asymmetric key agreement protocol for use in no-power, low-power, and IoT devices*; http://csrc.nist.gov/groups/ST/lwc-workshop2015/papers/session8-atkins-paper.pdf
- [Kyber] Joppe W. Bos, L. Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, J. Schanck, P. Schwabe, and D. Stehlé; 2017-06 *CRYSTALS-Kyber: A CCA-Secure Module-Lattice-Based KEM* https://pq-crystals.org/kyber , https://ia.cr/2017/634
- [NewHope] Erdem Alkim and Léo Ducas and Thomas Pöppelmann and Peter Schwabe; 2015-11 *CRYSTALS-Kyber: A CCA-Secure Module-Lattice-Based KEM* https://newhopecrypto.org , https://ia.cr/2015/1092
- [Saber] Jan-Pieter D'Anvers, Angshuman Karmakar, Sujoy Sinha Roy, and Frederik Vercauteren; 2017-06 *Saber: Module-LWR based key exchange, CPA-secure encryption and CCA-secure KEM* https://ia.cr/2018/230
- [SEC#1] *SEC 1: Elliptic Curve Crptography* http://secg.org
- [SIKE] David Jao, et al.; 2017 *SIKE – Supersingular Isogeny Key Encapsulation* http://sike.org
- [SQISign] Luca De Feo, David Kohel, Antonin Leroux, Christophe Petit and Benjamin Wesolowski; 2020-10 *SQISign: compact post-quantum signatures from quaternions and isogenies*; https://ia.cr/2020/1240

- [WalnutDSA] Iris Anshel, Derek Atkins, Dorian Goldfeld, and Paul E Gunnells; 2020-10 *WalnutDSA(TM): A Quantum-Resistant Digital Signature Algorithm*; https://ia.cr/2017/058
- [NN21] Daniel Nager, and Jianfang "Danny" Niu; 2020-10 *Xifrat - Compact Public-Key Cryptosystems based on Quasigroups*; https://ia.cr/2021/444