Xifrat – Compact Public-Key Cryptosystems based on Quasigroups

This paper proposes a new public-key cryptosystem based on a quasigroup with the special property of "restricted-commutativity". We argue its security empirically and present constructions for key exchange and digital signature. To the best of our knowledge, our primitive and construction have no known polynomial-time attack from quantum computers yet.

The proposed cryptosystem along with the reference implementation are released to the public domain.

Authors:
Daniel Nager (daniel.nager {at} gmail {dot} com),
"Danny" Niu Jianfang (dannyniu {at} hotmail {dot} com)

Table of Contents

Figures

1. Introduction

Public-key cryptography is a branch of modern cryptography that studies the protection of secrecy and authenticity in an open and public environment. Two most important functionalities of public-key cryptography are digital signature (authenticity) and public-key encryption and (its functional equivalent) key exchange (secrecy).

Two most prominent development occurred in the 1970s - the Diffie-Hellman key exchange [DH76], and the RSA cryptosystem [RSA78], with current cryptanalysis showing that at least 2048-bit parameters are required for 112-bit security level due to the "General Number Field Sieving" algorithm [LL93]. The most prominent industry standard for RSA is [PKCS#1].

The most compact public-key cryptosystems known today are based on the elliptic-curve discrete logarithm problem, with current cryptanalysis showing that only 256-bit parameters are needed for 128-bit security level due to Pollard's rho algorithm for discrete logarithm [Po78]. The most prominent industry standard for elliptic-curve cryptography would be [SEC#1].

The search for compact public-key cryptosystem would have stopped when Berstein et al. introduced Curve25519 for key exchange and Ed25519 for digital signature [Curve25519] [EdDSA], which offered the best-in-class efficiency, performance, and logical and practical security (that is, both mathematically correct and easy to implement correctly and free of side channels), if wasn't for there're polynomial-time *quantum computer* algorithms for integer factorization and discrete logarithm [Shor95].

For the list of finalist and alternate candidates in the 3rd round of the NIST Post-Quantum Cryptography project [NIST-PQC], ones with one compact cryptogram usually have another one that's very huge (compact public key for SPHINCS+ with huge signature, compact signature for Rainbow with huge public key), some with solid security records may have large cryptograms with no compact one at all (Classic McEliece). For the ones with overall acceptable cryptogram sizes (Dilithium, Falcon, Kyber, NTRU, Saber), those sizes are still larger than that of RSA by a factor of a single decimal digit, and larger than that of elliptic curve by an order of magnitude.

In this paper, we propose 2 compact cryptosystems - 1 for key exchange and 1 for digital signature - both based on a quasigroup with the special property of "restricted-commutativity". Although cryptosystems based on quasigroups had been proposed [GMK08] and broken [FOPG10] before, our construction is completely different from theirs. While we cannot disprove the existence of efficient algorithm that break our cryptosystems, we present empirical arguments for the security of our cryptosystems.

The proposed cryptosystem along with the reference implementation are released to the public domain.

2. Xifrat Elements - The Quasigroup

The core element of Xifrat cryptosystem is a quasigroup of 8 elements. We represent the binary operation of this group as $f : \mathbb{Z}_8 \times \mathbb{Z}_8 \rightarrow \mathbb{Z}_8$. This group has the following properties:

- Non-Associative *In General*: that is, for most cases, $f(f(a,b),c) \neq f(a,f(b,c))$
- Non-Commutative *In General*: that is, for most cases, $f(a,b) \neq f(b,a)$
- Restricted-Commutativity: that is, for all cases, $f(f(a,b),f(c,d)) = f(f(a,c),f(b,d))$

The table for the quasigroup is as follow:

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

The function $f(a,b)$ returns the value of the cell at a'th row and b'th column (indices are 0-based).

2.1. How did we choose such quasigroup

The first part is "why 8"? Actually, we originally had one of 16 elements; although it had all the desired property, its choice cannot be verified. So we went for one that can be verified as random. When we run our program to find a verifiably random quasigroup with 16 elements, it took so much time that we concluded that it's beyond the patience of any potential third-party verifiers, so we went for 8.

Second, we have a list of desired property of the quasigroup. In addition to the arithmetic ones listed above, we have the following:

- The quasigroup table should not have obvious symmetry;
- The quasigroup table should not have any fixed points;

With these requirements in mind, we created a simple C program that searched for candidate quasigroup tables. The program ran the following steps:

Figure 2.1. Quasigroup Search Program

1. Create an array of shuffles as follow:
    1. Seed an instance of SHAKE-256 XOF [FIPS-202] function with the NUL-terminated ASCII string: "`xifrat - public-key cryptosystem`"
    2. Create a list U of shuffles one by one using the following steps:
        1. `[label-1]`: Read 8 octets from the XOF stream and interpret it as a 64-bit little-endian unsigned integer.
        2. If the number is greater than $\text{floor}((2^{64}\text{-}1) / 8!) \cdot (8!)$ then go to `[label-1]` and proceed from there again.
        3. Take the number modulo $8!$ and label it as (i.e., set it to) s.
        4. Initialize the list V to be shuffled as [0,1,2 ... 7].
        5. For i in 8...2:
            1. $p \leftarrow s$ modulo i
            2. $j \leftarrow 8 - i$
            3. Swap V_{p+j} and V_j.
            4. $s \leftarrow \text{floor}(s / i)$
        6. Append V to U
2. Walk diagonally from top-right to bottom-right starting from the top-left corner to the bottom-right corner; the cells traversed in such pattern are labelled 0...63, and for each cell walked suchly:
    1. `[label-2]`: Set the value of the cell to that of the lowest index in the shuffle such that the constraints as set out and implied in the requirements are not violated.
    2. Recursively set the next cell similarly:
        1. If at some point constraints are violated, try the next index in the shuffle; and if all indices are tried out, recursively fix it by trying the next index in the previous cells by reverting to `[label-2]` . Until:
        2. When all cells are set and no constraint is violated, output the table and return [SUCCESS].

The source code for the program can be found at our GitHub repository: https://github.com/dannyniu/xifrat.

2.2. Miscellaneous information about the quasigroup

As we choose our quasigroup operation table in a verifiably random fashion, we should indicate how random the choice was. Below is "indices of freedom"

of our quasigroup, laid in walking order from left to right and top to bottom. The smaller the index indicates the more free the choice of the cell value is; a value of 9 indicates that the constraints from other cells had determined the value of that cell

```
1  2  1  6  4  7  3  4
7  2  3  1  1  3  5  9
9  9  9  9  9  9  9  9
9  9  9  9  9  9  9  9
9  9  9  9  9  9  9  9
9  9  9  9  9  9  9  9
9  9  9  9  9  9  9  9
9  9  9  9  9  9  9  9
```

Our program was originally written as a single-thread program. When it's found that more efficiency was more desired, we parallelized it with the `fork(2)` POSIX function, so that global states can still be shared by every function in the code, and minimal modification is needed.

3. Xifrat Primitive - The Mixing Function

The next building block in Xifrat, is the mixing function $c = m(r,k)$, where c, r, k are "cryptograms" of Xifrat. The function has the following property:

- Restricted-Commutativity, as is the case with $f(a,b)$
- Resistance to Key-Recovery: that is, given any number of pairs of c and r, it should be computationally infeasible to find k.
- Resistance to Cryptogram Prediction: that is, given c' or r' (but not both), and any number of such pair of cryptogram under a particular k, it should be computationally infeasible to find c' from r' and vice-versa.

3.1. The Construction and Instantiation of the Mixing Function

The mixing function $m()$ runs with 1 parameter: N - number of scalar elements from $\mathbb{Z}_8$ in the cryptogram vector. The scalar elements indexed i in a cryptogram p is p_i.

Figure 3.1. Algorithm for $m(r,k)$

1. For i in [0...N-1]:
   $r_i \leftarrow f(r_i\ ,\ k_i)$ // initial mixing of cryptograms r with k
2. Given $U \leftarrow \text{list}(\text{generator}(N{\cdot}N{+}1))$, for i in [0...N·N]:
   $r_i \leftarrow f(r_{U_i}\ ,\ r_{U_{i+1}})$ // mixing of r with itself in random sequence
   to block certain types of divide-and-conquer attack.

3. For i in [0...N-1]:
   $r_i \leftarrow f(r_i\ ,\ k_i)$ // final mixing of cryptograms r with k

---

In the above algorithm, the list() function takes the outputs of an iterator and assembles them in to an ordered list. The function generator() is defined as follow:

Figure 3.2. Algorithm for $generator(x)$

---

1. Create a list $U \leftarrow$ list($7*(3 \cdot i + 5)^{17} + 11 \bmod N$ for i in [0...N-1])
2. For i in [0...x-1]:
   1. yield $U_{i \bmod N}$
   2. If $(N - i \bmod N) = 1$:
      1. Copy U to V.
      2. $U_j \leftarrow V_{U_j}$ for j in [0...N]
   3. $i = i+1$
3. Return [END].

For ease of implementation, we choose a single parameter $N = 131$, resulting in a cryptogram size of 393 bits (rounded up to 50 octets), targeting 192-bit overall security. N is chosen to be a prime so that it's easy to build a random sequence generator that consumes few working memory.

As a cryptanalysis challenge proposed for the purpose of helping understand the properties of Xifrat cryptosystem better, we propose a toy parameter set N of 23 targeting 32-bit security.

3.2. The Arguments for Security

To argue for the security of $m()$, we observe that the middle loop of the mixing function is actually N rounds of application of Feistel network, with each **2 rounds** consisting of mixing of N tritets with each other permuted by the $f()$ function. Unlike regular Feistel network where operands are mixed sequentially, we randomized the order of mixing in order to deter potential 'divide-and-conquer' attacks.

Algebraically, each 8 values of the tritet needs to be represented as 64 × 64 sparse matrix - two 8 × 8 ones nested together, as an inner one and an outer one that receives operands differently depending on whether the operand is applied on the left or right. In other words, the arithmetic value of the matrix depends on whether it's used as row or column index when looking up in the sbox.

Next, each tritet output from the mixing function $m()$ is the product of appying $2N$ tritets - the r and k, where N is 131. We *roughly* estimate that potential algebraic attacks would either have to operate at very high degree,

or produce large set of linear relationships with too many derived linearized variables to be practical.

3.3. Proof of Correctness

To prove the correctness of "restricted-commutativity" of the $m()$ function, we'll first need a few propositions.

Notation 1. We simplify, both entry-wise application of scalar elements of cryptogram vectors, and direct application on tritet operands, $f(a,b)$ as ab , and $f(f(...(a,b),c...))$ as $abc...$; for a function $e()$ defined later, we denote $e(a)$ as a' .

Prop 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 **Prop 1.**

Definition 1. *$e(a)$ applies the middle loop (step 2) of the algorithm for $m(r,k)$ to a ; and per **Notation 1.** results in a' .*

Prop 2. $e(ab)=e(a)e(b)$

Proof: Due to the structure of the middle loop of $m()$, this is apparent by applying **Prop 1.**

Prop 3. $(a_1\ a_2\ a_3)(b_1\ b_2\ b_3)(c_1\ c_2\ c_3) = (a_1\ b_1\ c_1)(a_2\ b_2\ c_2)(a_3\ b_3\ c_3)$

Proof: From the left side of the equation, by **Prop 1.**, we have

$(a_1\ a_2\ a_3)(b_1\ b_2\ b_3)(c_1\ c_2\ c_3) =$

$(\ (a_1\ b_1)(a_2\ b_2)(a_3\ b_3)\ )\ (c_1\ c_2\ c_3) =$

$((a_1\ b_1)c_1)\ ((a_2\ b_2)c_2)\ ((a_3\ b_3)c_3) =$

$(a_1\ b_1\ c_1)(a_2\ b_2\ c_2)(a_3\ b_3\ c_3)$

Main Proposition $(ab)(cd) = (ac)(bd)$

Proof: observe that the $m(\mathrm{a},\mathrm{b})$ function can be represented using $e(x)$ as: $e(ab)b$, we first rewrite the left side of the equation as:

$(ab)(cd) = (e(ab)b)\ (e(cd)d) =$

$(e(a)e(b)b)\ (e(c)e(d)d) =$

$(a'\ b'\ b)\ (c'\ d'\ d) =$

$e(a'\ b'\ b)\ e(c'\ d'\ d)\ (c'\ d'\ d) =$

$(a''\ b''\ b')\ (c''\ d''\ d')\ (c'\ d'\ d) =$

Likewise for the right side

$(ac)(bd) =$

$(a''\ c''\ c')\ (b''\ d''\ d')\ (b'\ d'\ d) =$

By **Prop 3.**, the two expressions are equal.

4. Xifrat Schemes

For readability purposes, we inherit at here, the notations used in the previous section.

4.1. Xifrat-Kex the Key Exchange

Xifrat-Kex is a bi-party key exchange scheme similar to Diffie-Hellman [DH76]. Unlike recent lattice-based key encapsulation mechanisms or public-key encryption algorithms, Xifrat-Kex is "participant-symmetric" and has no "decryption failure".

Figure 4.1. The Xifrat-Kex Key Exchange Protocol

1. Agree on a public cryptogram C with the peer.
2. Generate a private key $K_{private}$, and send $K_{self} = (CK_{private})$ to the peer.
3. Receive the key share of the peer K_{peer}, and compute $K_{shared} = K_{peer}(K_{private}C)$.

Correctness: Suppose the private keys for oneself is K and for the peer is Q, the correctness of the key exchange is apparent from the restricted-commutativity property; the key exchange can be re-written as: $(CK)(QC) = (CQ)(KC) = K_{shared}$

4.2. Xifrat-Sign the Digital Signature Algorithm

Xifrat-Sign is a digital signature scheme consisting of 3 algorithms for key-pair generation, signature generation, and verfication. Xifrat-Sign uses a hash function, which is instantiated with the XOF SHAKE-256 by taking its first 384 bits of output.

Figure 4.2. Xifrat-Sign Key Generation

1. Generate 3 cryptograms: C, K, and Q.
2. Compute $P_1 = (CK)$, $P_2 = (KQ)$,
3. Return public-key $pk = (C,\ P_1,\ P_2)$, and private-key $sk = (C,\ K,\ Q)$,

Figure 4.3. Xifrat-Sign Signature Generation

1. **Input:** m - the message
2. Compute $\mathrm{SHAKE256}_{384}(m)$, and zero-extend the output to the size of Xifrat cryptogram, and obtain the result as H .
3. Compute $S = (HQ)$
4. Return S

Figure 4.4. Xifrat-Sign Signature Verification

1. **Input:** m - the message , S - the signature
2. Compute $\mathrm{SHAKE256}_{384}(m)$, and zero-extend the output to the size of Xifrat cryptogram, and obtain the result as H .
3. Compute $T_1 = P_1\ S$
4. Compute $T_2 = (CH)P_2$
5. If $T_1 = T_2$, return [VALID] ; otherwise, return [INVALID] .

Correctness: the formula for two verification transcripts: $T_1 = P_1\ S = (CK)(HQ)$ and $T_2 = (CH)\ P_2 = (CH)(KQ)$, by restricted commutativity, $T_1 = T_2$

Annex A. References

- [Curve25519] D.J.Bernstein, *Curve25519: new Diffie-Hellman speed records* https://cr.yp.to/papers.html#curve25519

- [DH76] W.Diffie, M.E.Hellman, *New Directions in Cryptography*. 1976 IEEE Transactions on Information Theory; Volume: 22, Issue 6, pp 644-654; https://doi.org/10.1109/TIT.1976.1055638
- [EdDSA] D.J.Bernstein, N.Duif, T.Lange, P.Schwabe, B.Yang *High-speed high-security signatures* https://cr.yp.to/papers.html#ed25519
- [FOPG10] J.Faugère, R.S.Ødegård, L.Perret, D.Gligoroski, *Analysis of the MQQ Public Key Cryptosystem* 2021 Cryptology and Network Security; Volume: 6467, pp169-183; https://doi.org/10.1007/978-3-642-17619-7_13
- [GMK08] D.Gligoroski, S.Markovski, S.J.Knapskog, *Public Key Block Cipher Based on Multivariate Quadratic Quasigroups* https://eprint.iacr.org/2008/320
- [LL93] A.K.Lenstra, H.W.Lenstra, *The Development of the Number Field Sieve*. Lecture Notes in Mathematics, vol. 1554, Berlin, Springer-Verlag, 1993.
- [Po78] J.M.Pollard, *Monte Carlo methods for index computation (mod p)*. Mathematics of Computations; Volume: 32, Issue 143, pp 918-924; https://doi.org/10.1090/S0025-5718-1978-0491431-9
- [RSA78] R.L.Rivest, A.Shamir, L.Adleman, *A method for obtaining digital signatures and public-key cryptosystems*. 1978 Communications of The ACM; Volume: 21, Issue: 2, pp 120-126; https://doi.org/10.1145/357980.358017
- [Shor95] P.Shor, *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer*. 1995 arXiv: Quantum Physics; https://doi.org/10.1137/S0097539795293172
- [FIPS-202] NIST FIPS-202 *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions* ; http://dx.doi.org/10.6028/NIST.FIPS.202
- [NIST-PQC] Post-Quantum Cryptography, Round 3 Submissions. https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions
- [PKCS#1] K.M.Moriarty, B.Kaliski, J.Jonsson, A.Rusch, *PKCS #1: RSA Cryptography Standard Version 2.2*. https://tools.ietf.org/html/rfc8017
- [SEC#1] *SEC 1: Elliptic Curve Crptography* http://secg.org