TBIR - Trapdoor Bijection from Integer Reciprocal.

1 This is the algorithm specification for TBIR - Trapdoor Bijection from Integer Reciprocal - an algorithm for instantiating key encapsulation mechanism and digital signature schemes.

Table of Contents

Figures

1. Algorithm Description.

2 TBIR stands for "Trapdoor Bijection from Integer Reciporcal". As the name suggests, it's a bijection, making it suitable for applications where a drop-in replacement for RSA is desirable, such as signature with message recovery. The mathematical components of TBIR consist of:

- Domain parameters:
  - K which is a finite field of order p.
    - For ease of notation, we define $p_{\text{bits}} = \text{floor}(\log_2(p))$ and $p_{\text{len}} = \text{ceil}(p_{\text{bits}} / 8)$
  - H which is a XOF of appropriate security level.
- input and output:
  - each is a pair (x_1, x_2) of integers from K.
- public key:
  - in its transport form, consist of
    - a fixed-length public seed (64-bytes for this revision of the proposal) which is a byte string.
    - 10 large integers, grouped into 3 tuples:
      - $E = (E_0, E_1, E_2, E_3)$.
      - $F = (F_0, F_1, F_2)$.
      - $G = (G_0, G_1, G_2)$.
  - in its operational form,
    - 4 large integers, expanded from the seed using the XOF
    - the same 10 large integers from the transport form.
- private key:
  - in its transport form,
    - two fixed-length seeds - 1 public, which is the same as that in the transport form of the public key; 1 private, used to derive every other non-seed components in the keys.
  - in its operational form,
    - 10 large integers, grouped into 3 tuples:
      - $A = (A_0, A_1 ; A_2, A_3)$.
      - $B = (B_0, B_1 ; B_2, B_3)$.
      - $C = (C_0, C_1)$.

3 The notation $(u, v ; w, x)$ represents row-major 2x2 square matrix, with *u* and *v* being on the upper row, and *w* and *x* on the lower row; also, *u* and *w* are on the left side of the matrix while *v* and *x* on the right.

1.1. Key Generation

4 A subroutine used by the overall key generation process known as `ExpandX` is defined as follow:

5 **ExpandX**:

- Input:
  - `n` : number of large integers to generate,
  - `seed` : the seed byte string to use,
- Output:
  - an array of `n` elements in K

1. initialize and empty array `ret` .
2. for `i` from 0 inclusive, up to but excluding `n` (i.e. zero-based indexing)
   1. compute byte string `u` as the output of XOF: `H(seed+bytes([i+0x30])).digest(p_len)` - that is, interpret `i` as an ASCII digit, append it to seed, hash it with XOF and obtain a p_{len} -byte digest.
   2. interpret `u` as an integer `v` in big-endian byte order (i.e. most-significant byte first).
   3. discard (clear) bits above bit p_{bits} to obtain an element in K
   4. append `v` to `ret` .
3. return `ret` .

6 The key generation process takes an RNG to produce the public and private seeds, and computes the key components which are a set of large integers.

7 **TBIR-KeyGen**:

- Input:
  - an implicit instance of random number generator.
- Output:
  - `pk` : the public key,
  - `sk` : the private key.

1. Generate `pkseed` and `skseed` as 2 different fixed-length byte strings.
2. Compute E = ExpandX(4, pkseed) .
3. Compute A = ExpandX(4, skseed + "a") , that is, append the ASCII lower case letter `a` to `skseed` and use that to expand 4 large integers.
4. Compute C = ExpandX(2, skseed + "c") similarly, *but* only 2 components.
5. Compute the modular multiplicative inverse d of the determinant of row-major matrix A .
6. Compute the determinant B_0 of matrix $(E_0, E_1 ; A_0, A_1)$,

7. Compute the determinant B_1 of matrix $(E_0, E_1 ; A_2, A_3)$,

8. Compute the determinant B_2 of matrix $(E_2, E_3 ; A_0, A_1)$,

9. Compute the determinant B_3 of matrix $(E_2, E_3 ; A_2, A_3)$,

10. Produce the matrix $B = (dB_0, dB_1 ; dB_2, dB_3)$.

11. Compute a vector $S = (A_0 A_2 , A_0 A_3 + A_1 A_2 , A_1 A_3)$.

12. Compute a tuple F from vector scalar-multiplication $S \cdot C_0$.

13. Compute a tuple G from vector scalar-multiplication $S \cdot C_1$.

14. Return pk = (pkseed , E , F , G) , sk = (skseed , A , B , C) .

1.2. Public and Private Functions

8 The public forward function, otherwise known as '*Enc*' is defined as follow:

9 **TBIR-Enc**:

- Input:
  - a tuple of 2 elements $x = (x_0 , x_1)$
- Output:
  - a tuple of 2 elements.

1. Compute $U = E \cdot x$.

2. Compute a vector $w = (x_0^2 , x_0 x_1 , x_1^2)$

3. Compute $V_0 = F \cdot w$ as a dot product.

4. Compute $V_1 = G \cdot w$ as another dot product.

5. Element-wise multiply entries in V with modular multiplicative inverses of entries in U , and return the result as a tuple. Assert that it has exactly 2 elements.

10 The private inverse function, otherwise known as '*Dec*' is defined as follow:

11 **TBIR-Dec**:

- Input:
  - a tuple of 2 elements $x = (x_0 , x_1)$
- Output:
  - a tuple of 2 elements.

1. Set M to element-wise product of modular multiplicative inverse of entries in x and entries in C .

2. Obtain the solution R of the linear system $M = B \cdot R$.

3. Set T to element-wise modular multiplicative inverse of entries in R .

4. Return the solution Y of the linear system $T = A \cdot Y$

1.3. Parameter Sets

- For 128-bit classical security, $p = 2^{255} - 19$ is recommended, along with the XOF SHAKE-128, or something of equivalent security.

  This yields a public key size of $32 \cdot 10 + 64 = 384$ bytes, and a cipher transcript size of $32 \cdot 2 = 64$ bytes.

- For 256-bit classical security, the Mersenne prime $p = 2^{521} - 1$ is recommended, along SHAKE-256, or something of equivalent security.

  This yields a public key size of $65 \cdot 10 + 64 = 714$ bytes, and a cipher transcript size of $65 \cdot 2 = 130$ bytes.

- For 512-bit classical security, the Mersonne prime $p = 2^{1279} - 1$ is recommended, along with a XOF providing appropriate security. In the absence of existing instances, a non-standard SHAKE-512 is recommended where the internal structure is the same as SHAKE-256, except the capacity of the sponge is increased to 1024 bits.

  This yields a public key size of $160 \cdot 10 + 64 = 1664$ bytes, and a cipher transcript size of $160 \cdot 2 = 320$ bytes.

2. Design Rationale.

Figure 2.1. Algorithmic Structure

```
     c1             c2
 ----------     ----------
  b1     b2      b3     b4
-----  -----   -----  -----
a1 a2  a3 a4   a1 a2  a3 a4
x1 x2  x1 x2   x1 x2  x1 x2
```

12 TBIR consist of a 3-round substitution-permutation network (SPN). The substitution is a simple modular multiplicative inverse, this require that the set of elements form a field. The permutation is simply multiplication of 2-component vector with 2x2 matrix.

13 A naive way to produce the components required for the operation of TBIR, would be to generate every individual A, B, and C, then compute their composition, and output that as the public key. Inspired by the SNOVA MQ-based digital signature scheme designed by a team of scholars from Chinese Taipei, the author asks the question: can we generate a large portion of the public key from a seed, then compute much of the private key components *backwards*?

14 Therefore, we compute the 2nd round public coefficients E from the public seed `pkseed`. Then, we derive the 1st round coefficients A from the *private* seed (with domain separation of course) and the 2nd round private coefficients B from E and A. The 3rd round coefficients have to be computed and transported verbatim, so it cannot be compressed or expanded from a seed.

15 Both A and B are permutation matrices, applied to the input tuple. The components of C on the other hand, is used to mask away the products of elements of A, and are multiplied with the input tuple to produce the cipher transcript output from the private function.

2.1. Hardness Assumption

16 Because the resulting bijection redueces to a pair of bivariate quadratic equations over finite field, the difficulty of producing a solution of such equation system is the basis for the security of TBIR.

17 As of Oct. 2025, the author is not aware of any algorithm solving the equation system that exploits its structure (i.e. not aware of any solution other than brutal-forcing it). There are however, efficient algorithms for deciding whether such equation system has any solution. The author is uncertain whether this poses a threat to the secutiy of TBIR.

2.2. Parameter Sets

18 The selection of XOFs for the scheme is non-controvercial. The choice of the size of the prime is to twice the level of security intended to provide - for one, it prepares for the anticipation of unknown attacks such as birthday probability collision, and other reason is to increase difficulty from quantum cryptanalysis.

19 The prime numbers are chosen to be the same as that for Curve25519 and P-521 so that existing implementations on these algorithms can be repurposed for implementing TBIR. The Mersenne primes of magnitude 521 and 1279 are chosen because hashing to these fields result in very low chance of overflowing, which is why the prime numbers for P-256 or SM2 were not chosen.

2.3. Other Discussions

20 As evidenced numerous times elsewhere, defining algorithms in terms of bits causes interoperability troubles with byte-oriented programming, not to mention there's already the issue of endianness in the world of bytes.

21 The generation and encoding of numbers therefore are defined in terms of bytes in big-endian byte order. The **ExpandX** algorithm don't expand bits for this very reason, more so because the field sizes are not integral multiplies of 8, so shifting bits would be an implementation nightmare, and masking is used instead to discard unused bits.

22 The components of this cryptosystem are supposed to be byte-packed, and for components that're bit strings, they're zero-padded to byte boundaries. ASN.1 syntax shouldn't be used below application level at all in this algorithm.

3. Security Analysis.

23 The particular key establishment and digital signature schemes TBIR-KEM and TBIR-FDH inherit their security proof from random oracle model.

24 The way private components are multiplied and added together ensures that private key recovery from public key is impossible. Therefore the remaining attack(s) to consider would be evaluation of the private function without the private key. To his end, an adversary can rewrite the public function as a system of 2 bivariate quadratic equations over finite field and attempt to obtain its solution. As of Oct. 2025, the author is not aware of methods for obtaining such solution other than trying all possible solutions - i.e. brutal-force.

25 One major side channel with this scheme is with the computation of modular multiplicative inverses. While major cryptography libraries on the market such as OpenSSL may have done extensive optimization over this, care should nonetheless be taken when refactoring old or creating new codes.

4. Performance Evaluations.

5. Features

26 TBIR is among the most compact post-quantum cryptosystems in terms of transmission bandwidth - at 512-bit security, a public key is about 1664 bytes - on par with ML-DSA-44 which provides 128-bit classical security.

27 TBIR is also a simple scheme to comprehend, leading to less confusion with implementations, thus enhancing practical security.

28 TBIR is novel in its use of special technique to generate large part of the public key from seed, and compute large part of private key components from the public key. TBIR is also novel for being a bijection that enables the implementation of functionalities such as signature with message recovery.