The AJPS Cryptosystem

Aggarwal, Joux, Prakash & Santha

Presented by Team HKV: Henry Herzfeld, Michael Keller, Yuri Villanueva

Outline

- Motivation
- Background information: Mersenne primes, Hamming weights and distances
- The hard problem on which the cryptosystem is based
- How AJPS works
- Known attacks
- Our implementations
- Attacks that we tried
- Conclusions

Motivation

- Simple
- Easier to understand
- Authors believe that it's secure against quantum attacks
- Previous attempt by MK unsuccessful

Mersenne Primes

Mersenne number is

$p = 2^n - 1$

$p = 0b11111...11111$

when n is prime.

p is a Mersenne *prime* when p itself is prime.

The finite field $\mathbb{Z}_p$ when p is a Mersenne prime

- $\mathbb{Z}_p$ is a finite field when p is prime.
- When p is a Mersenne prime, $\mathbb{Z}_p$ has these nice properties:
  - 0b11111...11111 ≡ 0 (mod p)
  - Multiplication by 2 is a circular bit shift

Hamming Weight and Hamming Distance

- The Hamming weight of an integer m, written Ham(m), is the number of 1s in its binary representation.
- The Hamming distance between two integers m and n is Ham($m \oplus n$).
- Hamming weight properties:
  - $\mathrm{Ham}(x + y) \leq \mathrm{Ham}(x) + \mathrm{Ham}(y)$
  - $\mathrm{Ham}(xy) \leq \mathrm{Ham}(x) \cdot \mathrm{Ham}(y)$
- Additionally, in $\mathbb{Z}_p$, with $p = 2^n - 1$, a Mersenne prime, we have the following nice properties:
  - For all i, $\mathrm{Ham}(2^i x) = \mathrm{Ham}(x)$, because multiplication by 2^i in $\mathbb{Z}_p$ is just a cyclic bit shift.
  - $\mathrm{Ham}(-x) = n - \mathrm{Ham}(x)$ for all nonzero x in $\mathbb{Z}_p$. This is because 0b11111...11111 $\equiv 0 \pmod{p}$

Security Assumptions

That is, on what hard problem is the AJPS cryptosystem based?

- Bit-by-bit- Mersenne Low Hamming Weight Problem
- The *Mersenne Low Hamming Combination Assumption:* Given an n-bit Mersenne prime and an integer h, such that $4h^2 < n \leq 16h^2$, the advantage of a probabilistic polynomial-time adversary in distinguishing

$$\left(\begin{bmatrix} R_1 \\ R_2 \end{bmatrix}, \begin{bmatrix} R_1 \\ R_2 \end{bmatrix} \cdot A + \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}\right) \quad \text{and} \quad \left(\begin{bmatrix} R_1 \\ R_2 \end{bmatrix}, \begin{bmatrix} R_3 \\ R_4 \end{bmatrix}\right)$$

is at most $O(2^{-h})$, where R_1, R_2, R_3, and R_4 are independent and uniformly chosen random n-bit strings, and A and B are independently-chosen n-bit strings of Hamming weight h.

AJPS Single bit Encryption: Keygen

- Given the security parameter λ, choose a Mersenne prime $p = 2^n - 1$ and an integer h such that $\binom{n}{h} \geq 2^\lambda$ and $4\,h^2 < n \leq 16\,h^2$.
- Choose F, G to be two independent n-bit strings chosen uniformly at random from all n-bit strings of Hamming weight h.
- Set $\mathsf{pk} := H = \mathrm{seq}(\frac{\mathrm{int}(F)}{\mathrm{int}(G)})$, and $\mathsf{sk} := G$.

AJPS Single Bit Encryption

Encryption. The encryption algorithm chooses two independent strings A, B uniformly at random from all strings with Hamming weight h. A bit b is encrypted as

$$C = \mathsf{Enc}(\mathsf{pk}, b) := (-1)^b (A \cdot H + B) \ .$$

AJPS Single Bit Decryption

Decryption. The decryption algorithm computes $d = \mathsf{Ham}(C \cdot G)$. If $d \leq 2h^2$, then output 0; if $d \geq n - 2h^2$, then output 1. Else output $\perp$.

For the correctness of the decryption note that $C \cdot G = (-1)^b \cdot (A \cdot F + B \cdot G)$ which, has Hamming weight at most $2h^2$ if $b = 0$, and at least $n - 2h^2$ if $b = 1$.

Inefficient for large ciphertexts- multi bit variant required

AJPS Multi-bit Encryption: Keygen

1. Choose a uniformly random n-bit integer R in $\mathbb{Z}_p$.
2. Choose random F and G among integers in $\mathbb{Z}_p$ such that their binary representation has low Hamming weight h.
3. Compute $T = F \cdot R + G \pmod{p}$
4. The public key is $pk = (R, T)$.
5. The private key is $sk = F$.

AJPS Multi-bit Encryption: Encrypt

Encrypt with the public key (R, T)

1. The scheme needs an efficient error-correcting code, with encoding function $\mathcal{E} : \{0, 1\}^n \rightarrow \{0, 1\}^k$ and decoding function $\mathcal{D} : \{0, 1\}^k \rightarrow \{0, 1\}^n$
2. To encrypt a message $m \in \{0, 1\}^n$, select three random numbers A, B_1, and B_2 of low Hamming weight m.
3. Output the ciphertext (C_1, C_2), where

$$C_1 = A \cdot R + B_1 \text{ and}$$

$$C_2 = (A \cdot T + B_2) \oplus \mathcal{E}(m)$$

AJPS Multi-bit Encryption: Decrypt

Decrypt $C = (C_1, C_2)$ with the secret key F by computing the output $\mathscr{D}(C_2 \oplus C_1 \cdot F)$

How does it work?

Recall that $C = (C_1, C_2)$ with $C_1 = A \cdot R + B_1$ and $C_2 = (A \cdot T + B_2) \oplus \mathcal{E}(m)$.

$C_1 \cdot F = (A \cdot R + B_1)F = ARF + B_1 \cdot F \quad = A \cdot T + B_1 \cdot F$

Compare this with $C_2 = (A \cdot T + B_2) \oplus \mathcal{E}(m)$.

Since B_1, B_2, and F have low Hamming weight h, the hamming distance between $A \cdot T + B_1 F$ and $A \cdot T + B_2$ is low, allowing them to almost cancel each other out with $\oplus$, leaving the a result that has low Hamming distance from $\mathcal{E}(m)$.

Error-Correcting Code

- Error is random and spread out, so we can get away with a simple and efficient encoding.
- A simple repetition code works: k is the maximum message length in bits. ϱ is the number of repetitions of a single bit.
- Example: for message m = 0b1010, $\mathcal{E}(m)$ = 0b111...1000...0111...1000..0.
- The decoder $\mathscr{D}(x)$ looks at blocks of ϱ bits and decodes each block to a single bit, 0 or 1 depending on which value is the majority in the block.
- The recommended parameters: n = 756,839, k = 256, and ϱ = 2048.

Random Oracle Implementation

```
m1,m2,m3 = {}, {}, {}

A = oracle(K, n, h, m1)
B1 = oracle(K, n, h, m2)
B2 = oracle(K, n, h, m3)
```

Memoization of returned pseudorandoms

Each memo (dictionary) defines a unique oracle

PRNG can also be constructed into oracle via explicitly defining seed prior to PR generation

```
def oracle(x, n, h, memo):
    if x not in memo.keys():
        memo[x] = get_nbit_ham_strings(n, h, 1).pop()
    return memo[x]
```

Key Encapsulation Mechanism (KEM)

Random Oracles

Why do we need it?

H_1, H_2, H_3 are ROs

Key Encapsulation. Given the public key $\mathsf{pk} = (R, T)$, the algorithm Encaps proceeds as follows:

1. Pick a uniformly random λ-bit string K.
2. Let $A = \mathcal{H}_1(K)$, $B_1 = \mathcal{H}_2(K)$, and $B_2 = \mathcal{H}_3(K)$.
3. Let $C = (C_1, C_2)$, where $C_1 = A \cdot R + B_1$, and $C_2 = \mathcal{E}(K) \oplus (A \cdot T + B_2)$.
4. Output C, K.

Decapsulation. Given a ciphertext $C = (C_1, C_2)$, and $\mathsf{sk} = F$, the decapsulation algorithm Decaps algorithm proceeds as follows:

1. Compute $K' = \mathcal{D}((F \cdot C_1) \oplus C_2)$.
2. Let $A' = \mathcal{H}_1(K')$, $B'_1 = \mathcal{H}_2(K')$, and $B_2 = \mathcal{H}_3(K')$.
3. Let $C' = (C'_1, C'_2)$, where $C'_1 = A' \cdot R + B'_1$, and $C'_2 = \mathcal{E}(K') \oplus (A' \cdot T + B'_2)$.
4. If $C = C'$, output K', else output $\perp$.

Previous Attacks

N must be prime

Aggarwal et al. [1] propose several potential attacks on their system. We will summarize these here.

Weak key attack. Originally propsed by Beunardeau et al. [2], this attack is only relevant if all the active bits of F and G are in the less significant half of the string, that is, the right half of the string. If this is true, then F and G are smaller than $\sqrt{P}$ and thus they can be easily recovered by a continued fraction expansion of H/P.

LLL lattice attack. Also proposed by Beunardeau et al. [2], this generalizes the weak key attack by guessing a decomposition of F and G into windows of bits so that all the active bits are on the right. By replacing the continued fraction method with LLL, it is possible to recover F from any possible window decomposition. Aggarwal et. al [1] revised their security parameter λ to 256 to counter this.

Quantum attack using Grover's algorithm. Using Grover's algorithm [4] for a quantum computer, one can speed up the lattice attack by a quadratic factor. Thus, we must ensure that our security parameter λ and h are equal.

Meet in the middle attack. This attack, proposed by de Boer et al. [3], has no effect on the security level of this cryptosystem for the chosen parameters, as its complexity is much larger than 2^h.

Active attacks. Attacks of this type use the decryption of incorrectly encrypted ciphertexts to recover information about the secret key. To apply them to this cryptosystem, assume we have access to a decryption oracle. It is theoretically possible to leak information about the secret key by forming pseudo ciphertexts of the form $A * H + B*$ with low Hamming weights that are not λ. To counter this, we can use the key encapsulation and decapsulation algorithms described in Section 1.

Attack if *p* is not Prime

If n_0|n ,then q=2^n_0-1 divides p=2^n-1. F,G have Hamming weight <=h mod q.

Y=FR+G (mod q). We can try to guess G from this, in time sqrt(n_0 choose h). This will reveal F mod q, allowing us to guess F and G much faster than if p is prime.

Our Implementations

We implemented AJPS three different ways:

- Representing large integers with sequences of bytes
- Using Python's built in large numbers, later replacing it with the GNU Multiprecision Arithmetic Library's ints.
- Representing low-Hamming-weight integers in $\mathbb{Z}_p$ with a list of the positions of the 1s in their binary expansion.

```
# Repetition encoding. k is max bits in message, rho is number of repeated bits

# k = maximum length of message in bits (256 bytes or UTF-8 characters)
# rho = the number of repetitions of a bit

def E(m, k = 2048, rho = 256):
    acc = 0
    block_of_ones = 2**rho - 1 # 11111...111
    while m:
        if m & 1:
            acc += block_of_ones
        block_of_ones <<= rho
        m >>= 1
    return acc


def D(m, k = 2048, rho = 256):
    acc = 0
    block_of_ones = 2**rho - 1 # used as a mask for bitwise and
    set_bit = 1
    majority = (rho // 2) + 1
    while m:
        if hamming_weight(m & block_of_ones) >= majority:
            acc |= set_bit
        m >>= rho
        set_bit <<= 1
    return acc
```

Using Python's built-in bignum and Bit Shifts

- At first, no special data structures were used--just Python's built-in bignums.
- Because of $\mathbb{Z}_p$'s special properties, we liberally used bit shifts and bitwise logical operators.
- It handles 756,839-bit numbers just fine.
- On a 2013 Mac Pro, it takes about 1.5s for keygen, and around 0.8s-1s for encrypting and decrypting a 44-byte message.
- Performance increases by over a factor of 10 with gmpy2.

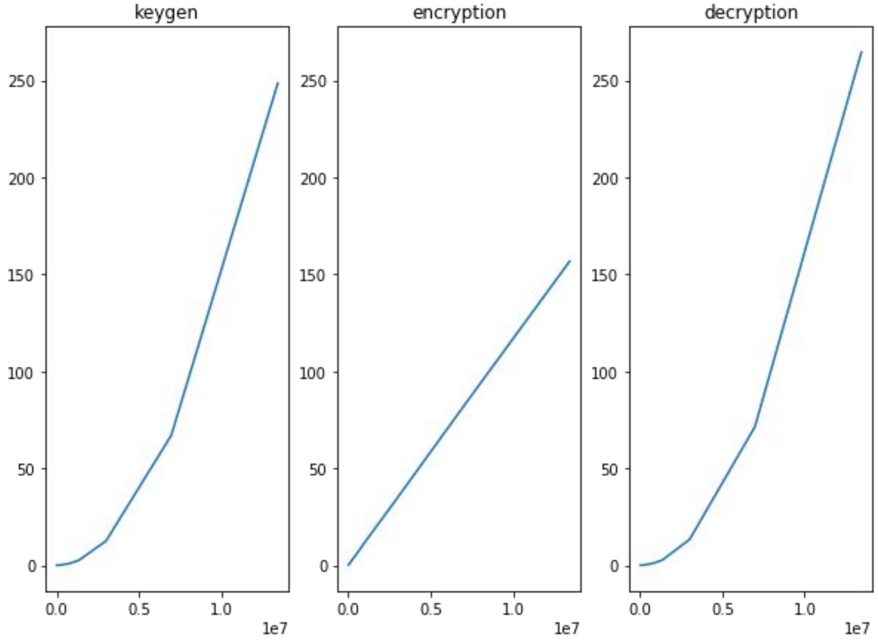
```
def get_random_int_of_hamming_weight_h(h, n):
    a = 0
    for i in range(h):
        bit_mask = 1 << randrange(n)
        while a & bit_mask:
            bit_mask = 1 << randrange(n)
        a = a | bit_mask
    return a


def hamming_weight(a):
    acc = 0
    while a:
        if a & 1: acc+= 1
        a >>= 1
    return acc


def hamming_distance(a, b):
    return hamming_weight(a ^ b)
```

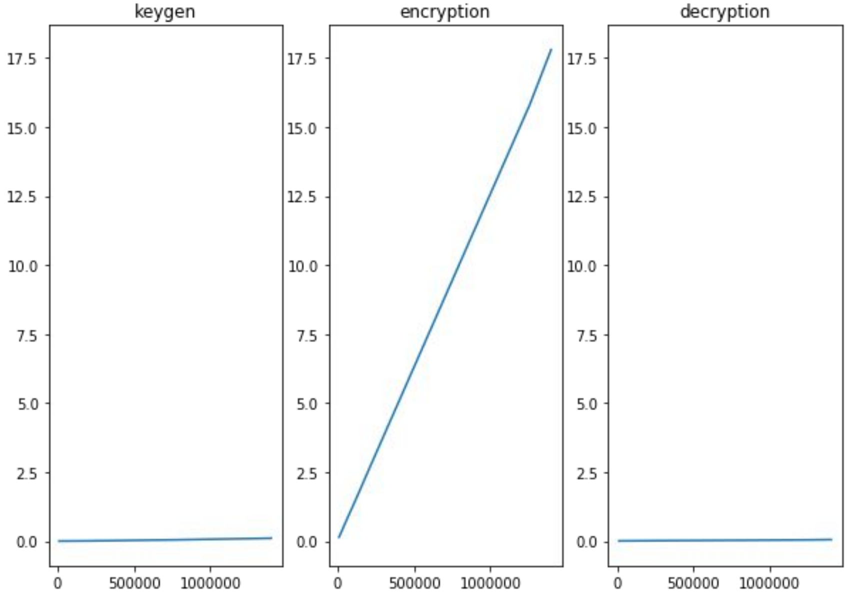
Using Bit Position Lists to save space

- With the recommended parameters, the secret key F is an integer up to 756,839 bits long, but has Hamming weight 256. This takes up almost 100 kilo*bytes* of space.
- Same goes for G, A, B_1, and B_2.
- If we represent these sparse integers as lists containing the positions of the 1s in their binary expansion, then each one takes up only 256 x 4 bytes = 1 kilobyte.
- Even less storage is needed if we use 20 bits per position—5120 bits, comparable to the length of an RSA key..
- Implementation using these bit-position lists is somewhat slower than the one with Python bignums, within a factor of 2.
- Performance also improves dramatically with gmpy2.

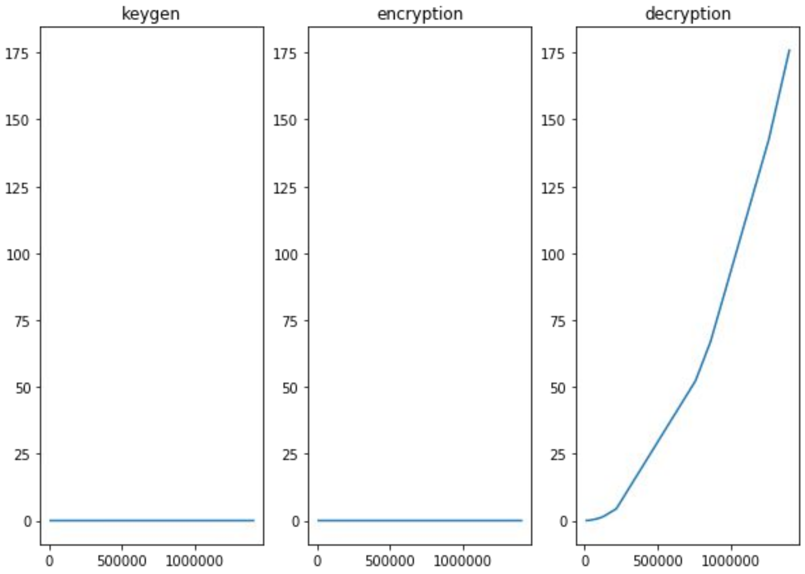
```python
def int_to_bit_position_list(a):
    acc = []
    i = 0
    while a:
        if a & 1: acc.append(i)
        i += 1
        a >>= 1
    return acc


def int_plus_bpl(a, bpl):
    return a + bit_position_list_to_int(bpl)


def int_times_bpl(a, bpl, pp = p):
    bpl.sort()
    acc = 0
    i = 0
    for bp in bpl:
        a = (a << (bp - i)) % pp
        acc = (acc + a) % pp
        i = bp
    return acc


def bit_position_list_to_int(bpl):
    return int_times_bpl(1, bpl)


def get_random_bpl_of_hamming_weight_h(h, n):
    bpl = list(map(int, np.random.choice(n, h, replace = False)))
    bpl.sort()
    return bpl
```

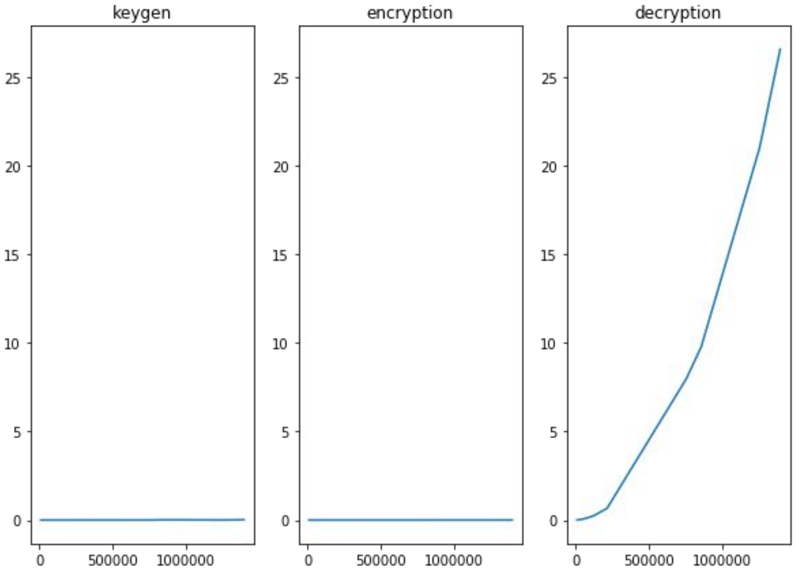
AJPS Full


keygen
encryption
decryption
250
200
150
100
50
0
0.0
0.5
1.0
1e7

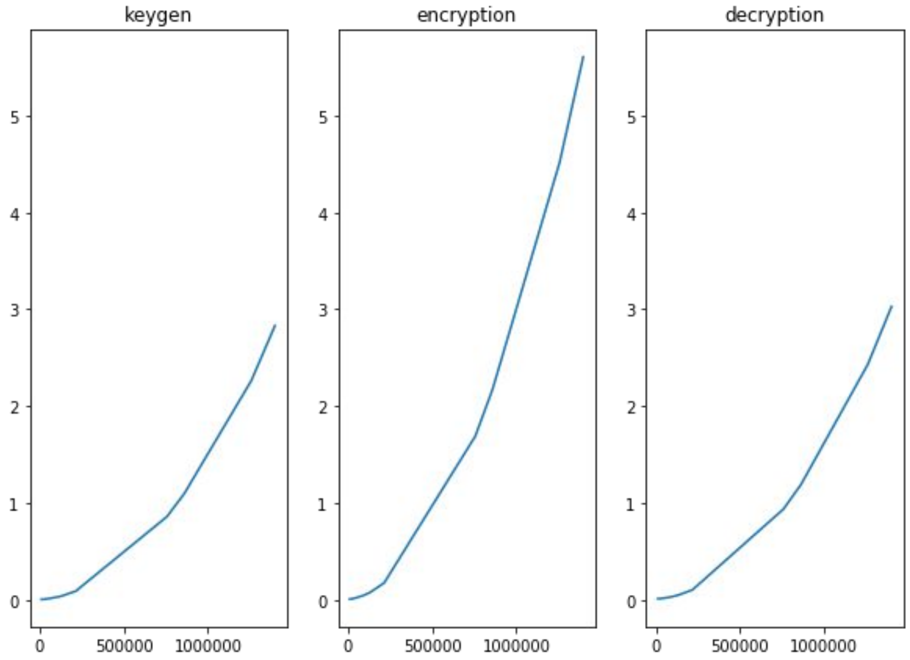

keygen
encryption
decryption
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0
500000
1000000

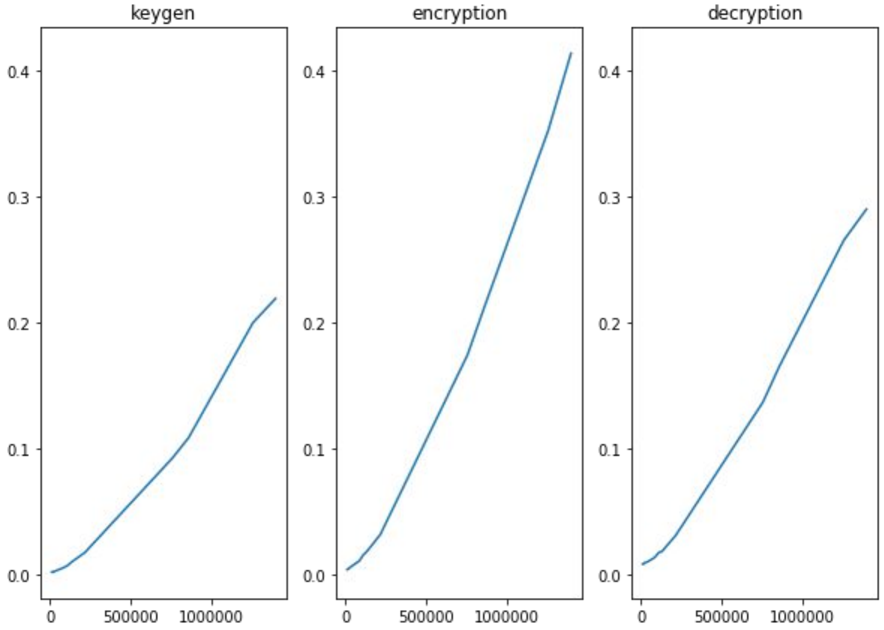
AJPS bit-by-bit





AJPS Full with Bit Position List





Our attempts at attacking AJPS

- Brute Force
- Meet in the middle (MITM)
- Physical attacks on the low-Hamming-weight noise

Brute Force

Search space is N choose W

Running time:

Classical: $O(n^{srt(n)/4})$

Quantum: $O(n^{sqrt(n)/8})$

```
pk, sk = keygen(n, h)

guess = get_nbit_ham_strings(n, h, 1).pop()

while ham(guess//pk) != h or guess//pk != sk:
    guess = get_nbit_ham_strings(n, h, 1).pop()
```

Upon announcement of AJPS, team claimed this was the optimal solution.

Meet-in-the-Middle Attack

Subset-sum problem

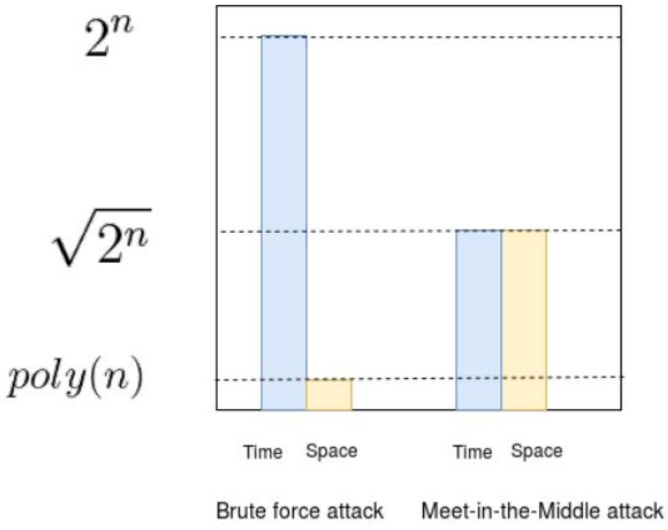
H = PK = F / G   (Bit-by-Bit)

Splitting guess for G into G_1, G_2

$hG_1 = - hG_2 + f$

Hash table L

Locality Sensitive hash function



Physical attacks on how these sparse integers are stored

- A little bit of noise is added to the public key and anything encrypted by AJPS:
  - Public key: $pk = (R, F \cdot R + G)$
  - Ciphertext: $(C_1, C_2) = (A \cdot R + B_1, (A \cdot T + B_2) \oplus \mathcal{E}(m))$
- If the attacker can corrupt the storage of this low-Hamming-weight noise, say wipe them out to zero or change them to something the attacker knows, they can recover the secret key F or the plaintext message m:
  - $pk = (R, F \cdot R + \cancel{G}) \Rightarrow$ Compute $R^{-1} \pmod p$, followed by $R^{-1}(F \cdot R + \cancel{G}) = F$
  - $(C_1, C_2) = (A \cdot R + \cancel{B_1}, (A \cdot T + B_2) \oplus \mathcal{E}(m)) \Rightarrow$ Compute $R^{-1} \pmod p$. The attacker can then recover $A = A \cdot R \cdot R^{-1}$, followed by $(A \cdot T) \oplus C_2 = (A \cdot T) \oplus (A \cdot T + B_2) \oplus \mathcal{E}(m)$.
- The job is even easier if we can wipe out A.
- What if we can make it so that $B_1 = B_2$?

References

[1] Divesh Aggarwal, Antoine Joux, Anupam Prakash, and Miklos Santha. A new public-key cryptosystem via Mersenne numbers. Cryptology ePrint Archive, Report 2017/481, version 20171206.004924, 2017.

[2] Marc Beunardeau, Aisling Connolly, Remi Geraud, and David Naccache. On the hardness of the Mersenne Low Hamming Ratio assumption. Technical report, Cryptology ePrint Archive, 2017/522, 2017.

[3] Koen de Boer, Leo Ducas, Stacey Jeffery, and Ronald de Wolf. Attacks on the AJPS Mersenne based cryptosystem. Cryptology ePrint Archive, Report 2017/1171, version 20180125.131924, 2018.

[4] Lov K. Grover. A fast quantum mechanical algorithm for database search. Proceedings of the twenty-eighth annual ACM symposium on Theory of computing, pages 212-219, 1996.