Leak the Secret Key of Elgamal Encryption in Cryptopp via Rowhammer

1 Analysing the Elgamal encryption scheme

Figure 1 shows the overview of an adversary and a victim in Elgamal's encryption scheme. The adversary ($\mathcal{A}$) injects faults into the victim's decryption oracle which uses sk as input. $\mathcal{A}$ sends a ciphertext (c) to the victim to get the result of their decryption query. Upon receiving the result, $\mathcal{A}$ uses Equation (7) to recover bits of the secret key. Different from the signature scheme where $\mathcal{A}$ can always get a faulty signature, the decryption query may not succeed because the victim may regard c as an invalid ciphertext.

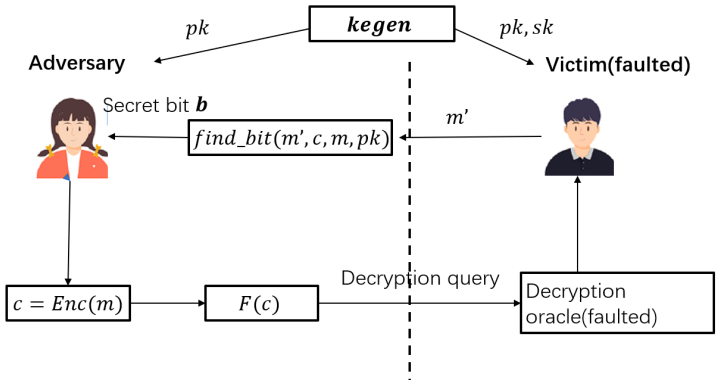

Figure 1: The overview of an adversary and a victim in Elgamal's encryption scheme. The adversary processes a ciphertext c via a function F and sends the result to the decryption oracle. The victim will return a plaintext m' to the adversary. F is a function that modifies the ciphertext based on Δx. If F computes the ciphetext like equation (7), the adversary can find bits of secret key via m'.

In the following, we first describe the Elgamal encryption scheme in Cryptopp, based on which, we show how to leak the secret key via Rowhammer.

$GenerateRandomWithKeySize(rng)$ is a function defined in Line 840 of *cryptopp/cryptlib.cpp*. This function reads a random generator rng and generate the public key pk and secret key sk. Particularly, pk can be viewed as: (q is the order of the group $\mathbb{G}$, g is the group generator and h is the exponentiate of g using secret key x)

$$pk = \{\mathbb{G}, q, g, h\} \tag{1}$$

For sk, it is defined as: (x is a number related to g, h.)

$$sk = \{x\} \tag{2}$$

pk and sk satisfy the following equations:

$$h = g^x \tag{3}$$

$SymmetricEncrypt(pk, m, rng)$ is implemented in Line 64 of *cryptpp/elgamal.h*. This is a function that takes pk, m and rng as inputs, where m is an encoded message. The function chooses a uniform y using rng and generates a ciphertext c as follows:

$$c = (c_1, c_2) = (g^y, h^y \cdot m) \tag{4}$$

$SymmetricDecrypt(c, pk, sk)$ is implemented in Line 83 of *cryptopp/elgamal.h* that takes a ciphertext c and pk, sk as inputs, and decrypts the ciphertext to plaintext:

$$m = decode(c_2/c_1^x) \tag{5}$$

When a single bit flip occurs to x right before $SymmetricDecrypt$ is invoked, the generated plaintext will become as follows:

$$m^{'} = decode(c_2/c_1^{x^{'}}) \tag{6}$$

where $m^{'}$ is a faulty plaintext, caused by a faulty secret key component $x^{'}$.

Here, we denote $x^{'}$ as $x + \Delta x$ where Δx represents the injected fault. To utilize Equation (5), we can guess a Δx, compute $F(c) = (c_1, c_2 \times c_1^{\Delta x})$ and check whether the following equation holds:

$$m = decode(c_2 \times c_1^{\Delta x}/c_1^{x^{'}}) \tag{7}$$

When Equation (7) holds, we are able to find out the index of the bit flipped in x and thus recover its original bit. To implement the bit recovery, a function called **find_bit** is proposed in Listing 2.1 in Section 2.1.

2 Recovering Secret Key via Rowhammer Fault

With the analysis above, we successfully simulated Rowhammer faults to get a total of 150 responses from Elgamal faulted decryption and each response was used to recover 1 unique bit out of the 512 secret bits.

2.1 Recovering Secret Bits

By inducing a single bit flip to a 512-bit secret key, we can get a response from a faulty victim decryption process. For the faulty response, we wrote a function called **find_bit** below to process it and recover the bit that has been flipped. Considering that only one bit is flipped for the secret key, we can enumerate all possible values for Δx, use i to indicate the index of a flipped bit and thus compute Δx_i as 2^i or -2^i. In Line 27 of our pseudo-code below, we start a loop to enumerate i. In Lines 36 to 45, Δx_i and the decryption query are used to check if index i is the correct index. We use a flag bit called **isValidCoding** to check if Equation (7) holds. If the check succeeds, we stop the loop and get index i and its corresponding Δx_i. As Rowhammer flips a bit either from 0 to 1 or from 1 to 0, i indicates which bit has been flipped and Δx_i indicates what its original bit value is. After we induce a bit flip for almost

every secret bit, we can recover the whole key.

```
#include <iostream>
#include <cryptopp/osrng.h>
#include <cryptopp/secblock.h>
#include <cryptopp/elgamal.h>
#include <cryptopp/cryptlib.h>
#include <cassert>
#include <cryptopp/pubkey.h>
using namespace CryptoPP;
using namespace std;
#define Bits_of_x 512
Integer sk;
int find_bit(
            ElGamal::Encryptor encryptor, //an object contains the publickey
            secByteBlock m, //m is the encoded message
            AutoSeededRandomPool rng;//random generator
            ){
    //encrypt first to get ciphertext
    size_t ecl = encryptor.CiphertextLength( m.size());
    secByteBlock ciphertext(ecl),recovered;
    encryptor.Encrypt(rng, m, m.size(), ciphertext);
    //prepare the original c1,c2 in element format
    DL_GroupParameters<T> &params = encryptor->GetAbstractGroupParameters();
    size_t elementSize = params.GetEncodedElementSize(true);
    Element c1=params.DecodeElement(ciphertext, true);
    Element c2=params.DecodeElement(ciphertext+ecl/2, true);
    Element p=params.GetGroupOrder();
    for(int i=0;i<Bits_of_d;i++){ //enumerate all possible index of a bit-
        flip fault
        //operate the bit difference to c2 according to Equation (7)
        Integer d;
        d.setbit(i,1);
        Element d2=a_exp_b_mod_c(c1,d,p+1);
        Element c2p=a_times_b_mod_c(c2,d2,p+1);
        params.DecodeElement(true,c2p,ciphertext+ecl/2);
        //If the modified cipher c1,c2 can be decrypted to m by the
            decryption query, the index of a flipped bit is targeted
        DecodingResult result = decryptor.Decrypt( rng, ciphertext,
            ciphertext.size(), recovered );
        if(result.isValidCoding){
                sk.set_bit(i,1);
                return 1;
            }
            else{
                Element c2p=a_times_b_mod_c(c2,d2.InverseMod(p+1),p+1);
                params.DecodeElement(true,c2p,ciphertext+ecl/2);
```

```
43                  DecodingResult result = decryptor.Decrypt( rng, ciphertext,
                        ciphertext.size(), recovered );
44                  if(result.isValidCoding){
45                      sk.set_bit(i,0);
46                      return 0;
47                  }
48
49              }
50      }
51      return -1;
52  }
```

In our simulation, we have collected 150 faulty response from the decryption query, based on which, the same number of unique bits from the 512 secret bits have been recovered and can be further used to infer remaining bits of the secret key sk.

3 Mitigation

An effective mitigation is *check after decryption*, which requires the secret owner to check if the result is a valid plaintext before releasing it. This mitigation has been adopted by WolfSSL and OpenSSL to fix similar vulnerabilities [2–4]. Specifically, we check if the secret key is faulted by checking whether $h = g^x$. If not, it means a secret key has been faulted and the process should abort.

References

[1] *CVE-2019-19962*. Available from MITRE. 2019.

[2] *EdDsa: check private value after sign*. 2024. URL: https://github.com/wolfSSL/wolfssl/commit/c8d0bb0bd8fcd3dd177ec04e9a659a006df51b73.

[3] *Openssl commit: Add a protection against fault attack on message v2*. 2018. URL: https://github.com/openssl/openssl/pull/7225/commits/02534c1ee3e84a1d6c59a887a67bd5ee81bcf6cf.

[4] *RSA Decryption: check private value after decryption*. 2024. URL: https://github.com/wolfSSL/wolfssl/commit/de4a6f9e00f6fbcaa7e20ed7bd89b5d50179e634.