Metrics for the AsiaCrypt16 Implementation

Vitalis Salis

Abstract

We have computed some metrics for the prototype implementation [2] of a mixnet based on the shuffle argument proposed by Fauzi et al [1]. The goal of these metrics is to identify aspects of the code that are slow and find suitable replacements for them.

1 Introduction

The prototype implementation of the mixnet proposed by Fauzi et al, produces multiple implementation difficulties. On implementations of cryptographic protocols it is typical to use C for your cryptographic computation. Yet the prototype is implemented using python, so it has to switch between python and C for its computations. This may be a bottleneck of the prototype and the reason some operations are slower than they should. Another reason may be that the underlying C cryptographic operations themselves are not efficient, and a different C implementation might improve matters. The two reasons are not exclusive, and one might compound the other. We start by investigating the first.

2 Metrics

Table 1 contains a list of metrics for the various operations of the prototype. Most of the time is taken by the prover and the verifier, as expected, because these have the most computations that produce a context switch between Python and C.

The time taken by each of these operations is linear, meaning that for 200 ciphertexts you just double the numbers on the table.

3 Context Switches

A context switch happens when a python program communicates with a C program for various cryptographic computations. The reasoning behind believing that a context switch may be the bottleneck of the application is that python needs to create a PyObject containing the data it wants to communicate, and C also needs to create a PyObject to return the result of the computations.

Metrics		
Operation	Short Description	Time per 100 voters
Initialization	Creates the elliptic Curve and private keys	364ms
Encryption	Encrypts the votes	674ms
Random Permutations	Creates random numbers	1ms
Proof	The shuffle	2085ms
Verification	Verification of the shuffle	2738ms
Decryption	Decrypts the votes	489ms

Table 1: Metrics

The prover has various steps. In order to validate our theory about context switches we measured each of these steps. Two of those steps, while having the same number of iterations, had a significant time difference. In particular, `step2a` below took 100ms, while `step3a` took 700ms.

```
def step2a(sigma, A1, randoms, g1_poly_zero, g1rho,
↪  g1_poly_squares):
    pi_1sp = []
    inverted_sigma = inverse_perm(sigma)
    for inv_i, ri, Ai1 in zip(inverted_sigma, randoms, A1):
        g1i_poly_sq = g1_poly_squares[inv_i]
        v = (2 * ri) * (Ai1 + g1_poly_zero) - (ri * ri) * g1rho +
↪  g1i_poly_sq
        pi_1sp.append(v)
    return pi_1sp

def step3a(sigma, ciphertexts, s_randoms, pk1, pk2):
    v1s_prime = []
    v2s_prime = []
    for perm_i, s_random in zip(sigma, s_randoms):
        (v1, v2) = ciphertexts[perm_i]
        v1s_prime.append(tuple_add(v1, enc(pk1, s_random[0],
↪  s_random[1], 0)))
        v2s_prime.append(tuple_add(v2, enc(pk2, s_random[0],
↪  s_random[1], 0)))
    return list(zip(v1s_prime, v2s_prime))
```

First we attributed the time difference to various calls to zip and to tuple creation. After removing all the calls to zip we didn't notice any significant difference. This seemed to validate the context switches theory, because the slower step contained more context switches per iteration.

But that's not the case. Using cProfile we identified the main reason behind this difference. The slower step does more multiplications on elliptic curve elements. While it is expected that multiplication will be slower than addition, the difference was enough to dismiss the context switches theory.

Multiplication on our elliptic curve elements takes 575ms per 300 multiplications, while addition takes 5ms for 400 additions. If the real problem were context switches, then the addition wouldn't have such a huge difference with the multiplication, because it has more operations hence more context switches.

4 Comparing bplib and libsnark

The prototype implementation uses the bplib[4] python module. bplib implements bilinear pairings on elliptic curves while also supporting elliptic curve operations using the openssl library.

Another implementation supporting elliptic curve computations and bilinear pairings is libsnark[5].

The common characteristics of these libraries are that they both use the Ate Pairing and they use windowed exponentiation for optimization purposes.

A key difference of these implementations is that bplib uses the curve Fp254BNb, while libsnark uses bn128 which is a patch on the Fp254BNb curve. Also, libsnark supports vectorized exponentiation which boosts up its performance.

In order to compare these two libraries and validate our theory that libsnark is faster than bplib, we created two different tests using bplib and libsnark on each one. The tests did multiplications (the bottleneck of the prototype) on both elliptic curve groups.

The results validated our theory. Multiplying elements on the G2 group using libsnark yielded a performance of 0.38s/1000 ciphertexts while using libsnark yielded 3.22s/1000. On G1 libsnark produced 0.13s/1000 while bplib produced 0.96s/1000 ciphertexts.

5 Solutions

Since the real bottleneck are the multiplications on G2 elements, the most obvious solution is to use optimizations on the multiplication process.

As mentioned, libsnark computes multiplications faster than our current implementation. Yet libsnark is written in C++ and we want to use a python module. A python wrapper for libsnark would be useful, for our needs and the open source community.

References

[1] Prastudu Fauzi, Helger Lipmaa, and Michal Zajac. A shuffle argument secure in the generic model.

[2] Panos Louridas, Dimitris Mitropoulos, Georgios Tsoukalas, and Georgios Korfiatis. Implementation of the shuffle argument.

[3] Stfan van der Walt, S. Chris Colbert, and Gal Varoquaux. The numpy array: A structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[4] George Danezis bplib: A bilinear pairing library for petlib.

[5] SCIPR Lab libsnark: a C++ library for zkSNARK proofs