FAST IMPLEMENTATION OF ELLIPTIC CURVE DIFFIE-HELLMAN

M.Gollub, C. Heiniger, T. Rubeli, H. Zhao

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

Describe in concise words what you do, why you do it (not necessarily in this order), and the main result. The abstract has to be self-contained and readable for a person in the general area. You should write the abstract last.

1. INTRODUCTION

Cryptography is a complex and widespread computational problem in modern computer science research and applications. To ensure the safety of the exchanging of information, encryption provides the core foundation of a secure information system. Among different cryptographic problems, key exchange is an essential topic for two distant users to be able to share a secret without meeting physically. Internet technology companies have been exploiting ways to have efficient yet high performance algorithms to fulfill the needs of exchanging keys in this age where network security has been concerned more than ever. Various key exchange tools have been generated. Elliptic Curve Cryptography (ECC) has shown its advantages over traditional RSA (Rivest, Shamir and Adleman). By using less key sizes, ECC can apply a more efficient method without reducing the secure level of the encryption, whereas RSA requires the involvement of larger primer number to reach the same goal [**?**].

As an important application domain of ECC, Elliptic Curve Diffie-Hellman (ECDH) key exchange has attracted researcher's attention due to its safety and efficiency. A fast implementation of ECDH is highly in demand, as the number of Internet users increases drastically with the growing needs for information security and the speed of computation. ECDH is widely used and there are some potential to be improved in its computational performance. The challenge lies in the complex computational models and its natural requires for large integers. In this paper, we propose a fast implementation of high performance of ECDH key exchange.By comparing with popular cryptography library we proved the feasibility of our improvement of our optimization methods.

Brown provides the backbone of this project by providing a general standard of Elliptic Curve Cryptography[1] and specifying a well-established deployment requirements. Malik's work regarding a particular implementation in a FPGA card [**?**] shows the possibility of performance oriented optimization. The use of projective coordinates, as the most efficient optimization methods during our implementation, was proven by Blake et al.[2] as an effective method for expensive field inversions. In this paper we combine multiply approaches of optimizations and achieved satisfactory performance improvement and execution speedup.

We give a thorough description of background in section 2, proposed optimization methods in section 3 and show our experimental results in section 4. In Section 5, we discuss and analyze our approach.

2. BACKGROUND: ELLIPTIC CURVE DIFFIE-HELLMAN

The purpose of this section is to familiarize the reader with the mathematical foundations of the Diffie-Hellman key exchange and which algorithms we used to achieve this task. This chapter follows a bottom-up approach.

Finite Field arithmetic [1, p. 3-4]. For elliptic curve cryptography one is mainly interested in the prime finite field $\mathbb{F}_p$ and the characteristic 2 finite field $\mathbb{F}_{2^m}$. In this paper only elliptic curves over $\mathbb{F}_p$ are discussed.

Montgomery multiplication.

Elliptic Curve over $\mathbb{F}_p$ [1, p. 6-7]. An elliptic curve is defined by the equation

$$y^2 \equiv x^3 + ax + b \pmod{p} \tag{1}$$

where p is an odd prime number and $a, b \in \mathbb{F}_p$ are the parameters of the curve. Furthermore a, b need to satisfy $4a^3 + 27b^2 \not\equiv 0$. The elliptic curve $E(\mathbb{F}_p)$ consists of the Points $P = (x, y)\ x, y \in \mathbb{F}_p$ satisfying (1). Additionally we introduce $\mathcal{O}$ the so called point at infinity. The addition of points (affine coordinates) is defined as follows [1]

1. $\mathcal{O} + \mathcal{O} = \mathcal{O}$
2. $(x, y) + \mathcal{O} = \mathcal{O} + (x, y) = (x, y) \quad \forall (x, y) \in E(\mathbb{F}_p)$

3. $(x, y) + (x, -y) = \mathcal{O} \quad \forall (x, y) \in E(\mathbb{F}_p)$

4. Assume $x_1 \neq x_2$. $(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$ is defined as $x_3 \equiv \lambda^2 - x_1 - x_2 \pmod p$, $y_3 \equiv \lambda(x_1 - x_3) - y_1 \pmod p$ and $\lambda \equiv \frac{y_2 - y_1}{x_2 - x_1} \pmod p$

5. $(x_1, y_1) + (x_1, y_1) = (x_3, y_3)$ is defined as $x_3 \equiv \lambda^2 - 2x_1 \pmod p$, $\lambda(x_1 - x_3) - y_1 \pmod p$ and $\lambda \equiv \frac{3x_1^2 + a}{2y_1} \pmod p$

Rule 4. describes how to add two different points whereas rule 5. explains how to double a point. The set of points satisfying (1) and $\mathcal{O}$ together with this addition form a commutative group.

Jacobian Coordinates [2, p. 59 - 60]. "In cases where field inversions are significantly more expensive than multiplications, it is efficient to implement projective coordinates" [2] In Jacobian coordinates a point is represented as a triplet (x, y, z) satisfying (2).

$$y^2 \equiv x^3 + axz^4 + bz^6 \pmod p \tag{2}$$

In this project Jacobian coordinates were implemented. For the definition of the rules see [2, p. 59-60]

Double-and-add method. In order to implement the Diffie-Hellman key exchange we need to calculate dP fast.
Input $P \in E(\mathbb{F}_p)$, $d \in \mathbb{N}$
Output: $d \cdot P \in E(\mathbb{F}_p)$

```
N <- P
Q <- O
for i from 0 to m do
  if d_i = 1 then
    Q <- point_add(Q, N)
  N <- point_double(N)
return Q
```

Listing 1. double-and-add method

where $d = d_0 + d_1 2 + ... + d_m 2^m \quad d_i \in \{0, 1\}$

Diffie Hellman key exchange [3, p. 170-171]. Alice and Bob want to establish a secret over a public channel. We assume that the elliptic curve parameter: a prime p, $a, b \in \mathbb{F}_p$ a point G with high order are publicly known.

1. Alice chooses a secret integer u, computes $G_u = uG$, and sends G_u to Bob.

2. Bob chooses a secret integer v, computes $G_v = vG$, and sends G_v to Alice.

3. Alice computes $uG_v = uvG$

4. Bob computes $vG_u = vuG$.

One can verify that $uG_v = uvG = vuG = vG_u$. An eavesdropper knows G, G_u, G_v and his goal is to calculate uvG. This is known as the Diffie-Hellman Problem and is assumed to be a hard problem.

Complexity.

Cost measure.

3. YOUR PROPOSED METHOD

Now comes the "beef" of the paper, where you explain what you did. Again, organize it in paragraphs with titles. As in every section you start with a very brief overview of the section.

For this class, explain all the optimizations you performed. This mean, you first very briefly explain the baseline implementation, then go through locality and other optimizations, and finally SSE (every project will be slightly different of course). Show or mention relevant analysis or assumptions. A few examples: 1) Profiling may lead you to optimize one part first; 2) bandwidth plus data transfer analysis may show that it is memory bound; 3) it may be too hard to implement the algorithm in full generality: make assumptions and state them (e.g., we assume n is divisible by 4; or, we consider only one type of input image); 4) explain how certain data accesses have poor locality. Generally, any type of analysis adds value to your work.

As important as the final results is to show that you took a structured, organized approach to the optimization and that you explain why you did what you did.

Mention and cite any external resources including library or other code.

Good visuals or even brief code snippets to illustrate what you did are good. Pasting large amounts of code to fill the space is not good.

4. EXPERIMENTAL RESULTS

To evaluate the optimization result, we test the implementation in a 64 bit Arch Linux machine with Intel CPU of Skylake i7-6600U CPU @ 3GHz. The compiler and arithmetic environment are as following:

- GCC 6.1.1 compiler
- 64 bit multiplication (mul, mulx): 1 op/cycle
- 64 bit addition/subtraction (add, sub): 4 op/cycle
- 64 bit addition with carry (adc, adcx, adox): 1 op/cycle
- Carry addition only: peak performance of 2 ops/cycle 6 Gops/s on 1 core
- Compile flag: -O3 -mavx2 -mbmi2 -madx

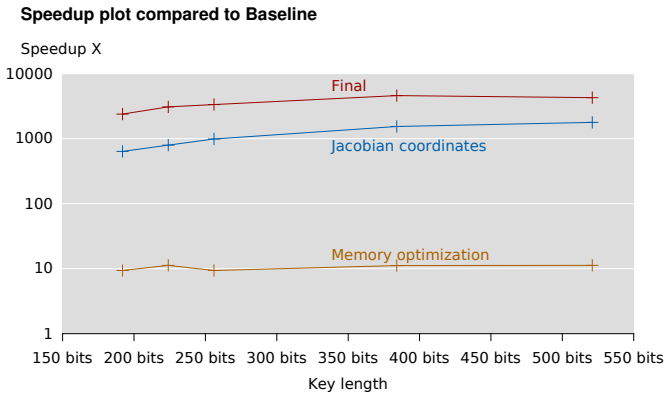

Fig. 1. Speedup plot compared to Baseline

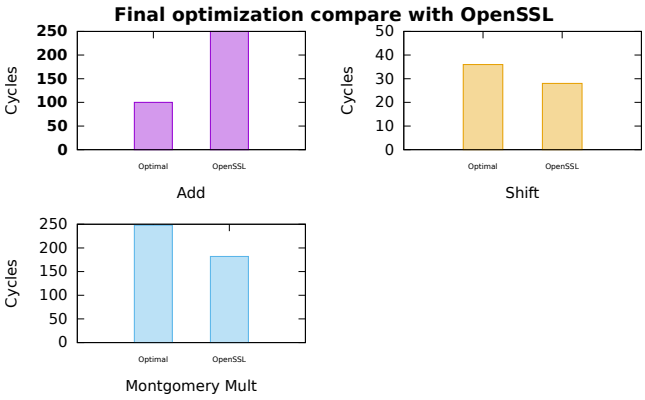

Fig. 3. Final optimization compared with OpenSSL

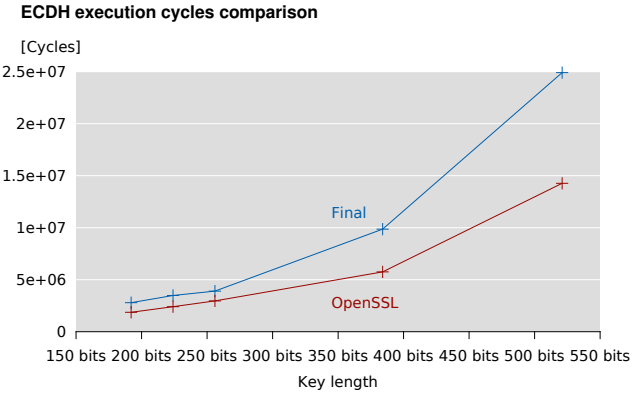

Fig. 2. ECDH execution cycles comparison

Based on the recommended Elliptic curve parameters from the guidance standard [1], we have chosen 5 predefined curves with various key size : 192, 224, 256, 384, 521 bits.All versions of implementation has been validated by using Google Unit test, by checking the key exchange result. The benchmark is done by calculating the CPU cycles, in order to testify the real performance by eliminating the influence of the machine. In the end we compare our implementation with OpenSSL, the well known open source library to check our implementation effect. The OpenSSL ECDH mechanism is deployed and the library has been imported as a package of GCC.

In the speedup comparison(Fig. 1) three crucial numbers are shown. It can be clearly seen that the Final implementation can execute the key exchange process 5000 times faster than the baseline in almost all the key lengths. In all three cases the speedup increases as the key size grows. It can be explained that the program is not fully functioning when the key size is relatively small, considering that the big integer operations(64 bits base) will take full operations even with smaller input size. The first milestone of memory optimization gives us about 10 times speedup, whereas Jacobian coordinates can run 1000 times faster. The speedup result is satisfactory and reasonable as the baseline implementation contains costly and expensive memory allocations.

Next we run the same operation with OpenSSL library and compare their computation cycles with us. It is no surprise that with the increasing of the key length the required cycles also increase rapidly(Fig. 2). Our required cycles remain the save level with OpenSSL with key length smaller than 256 bits, yet fall behind OpenSSL with larger key length.

Individual operations of integers are the main component of our algorithm. To see how good is our implementation we chose integer addition, shift and Montgomery multiplication to compare with OpenSSL(Fig. 3).This comparison is done with fixed key length of 192 bits. In the comparison of integer addition, our final implementation uses less cycles than OpenSSL big integer addition thus being faster than OpenSSL. Yet in both shift and Montgomery multiplication our implementation is more expensive and slower than OpenSSL.

Performance plots(Fig. 4 and Fig. 5) are consisted with the division of operation counts and cycles. It reflects if the implementation can reach up the machines' computation ability. Normal operations counts would stay constant as the optimization goes on. Yet in our case since the integer operations does not only include the normal integer operations, but also include the index counts. Thus we have to compromise by using code generated operation counts. In the following performance plot the operation counts changes, but it does reflect the computational efficiency of the code in certain level. In Fig. 4 the first comparison is done by showing the effect of memory optimization. It is clear that with huge amount of memory allocation, the baseline performance is really limited. In Fig. 5 we begin to see the real performance boost that comes from later optimization. The peak performance here is calculated by considering the ra-

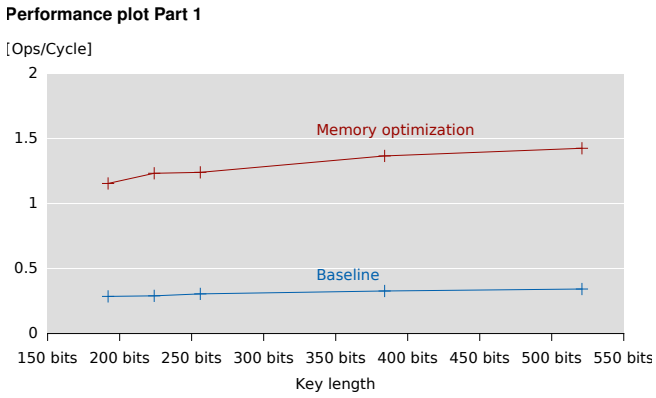

Fig. 4. Performance plot of baseline and memory optimization

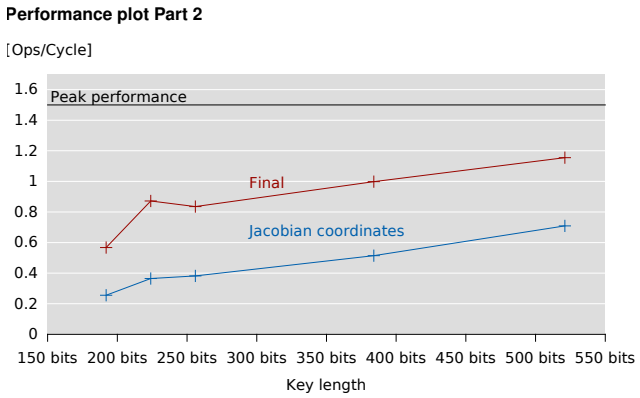

Fig. 5. Performance plot of Final optimization and Jacobian coordinates

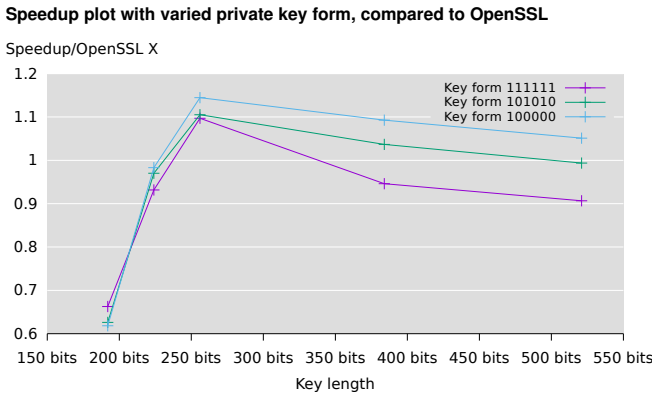

Fig. 6. Speedup plot with varied private key form, compared to OpenSSL

tio of additions and multiplications. With 1 add and 1 mult per cycle and two times add versus one mult, we can draw the line of peak performance at 1.5 ops/cycle. As we can see the final optimization are close to the peak performance with large key size, proving the optimization work successful.

Fig. 4 gives us another perspective of how different key form would influence the speedup. In the simplified private key form that consist of only zeros and ones, our speed of processing EDCH come closely to OpenSSL. The key form that consist only ones has the worst performance since it requires redundant integer operations that are unavoidable.

5. CONCLUSIONS

Here you need to briefly summarize what you did and why this is important. *Do not take the abstract* and put it in the past tense. Remember, now the reader has (hopefully) read the paper, so it is a very different situation from the abstract. Try to highlight important results and say the things you really want to get across (e.g., the results show that we are within 2x of the optimal performance ... Even though we only considered the DFT, our optimization techniques should be also applicable ....) You can also formulate next steps if you want. Be brief.

6. REFERENCES

[1] Daniel R. L. Brown, "Sec 1: Elliptic curve cryptography," http://www.secg.org/, 2009.

[2] I. Blake, G. Seroussi, and N. Smart, *Elliptic Curves in Cryptography*, Cambridge University Press, 1999, Cambridge Books Online.

[3] Lawrence C . Washington, *Elliptic Curves Number The-*

ory and Cryptography, Chapman and Hall/CRC, 2nd edition, 2008.