An FPGA Implementation of Elliptic Curve Cryptography for Future Secure Web Transaction

Jian Huang, Hao Li, and Phil Sweany
Department of Computer Science and Engineering
University of North Texas
Denton, TX, 76203

Abstract

Elliptic curve cryptography (ECC) is an alternative to traditional techniques for public key cryptography. It offers smaller key size without sacrificing security level. In a typical elliptic curve cryptosystem, elliptic curve point multiplication is the most computationally expensive component. So it would be more attractive to implement this unit using hardware than using software. In this paper, we propose an efficient FPGA implementation of the elliptic curve point multiplication in GF(2^{283}). We have designed and synthesized the elliptic curve point multiplication with Xilinx's FPGA. Experimental results demonstrate that the FPGA implementation can speedup the point multiplication by 31.6 times compared to a software based implementation.

1 Introduction

Cryptography is the most standard and efficient way to protect the security of web transactions. It can be used to protect the confidentiality, integrity, authentication, and non-reputation of the web transactions. There are two categories of cryptography schemes, i.e., public-key cryptography and symmetric-key cryptography. In public-key cryptography, the receiver and sender have their own private key and share a common public key. In symmetric-key cryptography, the receiver and sender must have the same private key, which makes it difficult to manage the private key. Public-key cryptography is easy for key distribution and key management. But it is not as efficient as symmetric-key cryptography [3]. Therefore, it is necessary to use dedicated hardware for public-key cryptography to improve the performance.

A well-known public-key cryptography algorithm is RSA, which was first proposed by Rivest, Shamir and Adleman in 1977 [4]. The security of RSA is based on hardness of integer factorization problem. It is commonly used in the secure sockets layer (SSL) protocol, which is the most popular way of protecting secure web transactions nowadays. SSL runs over transportation layer and it secures many application protocols such as HTTP, Telnet and FTP. However, due to the performance issue of RSA, using SSL usually slows down the web servers by three to nine times [5]. Elliptic Curve Cryptography (ECC) is a substitution for RSA which is very efficient. It was originally proposed by Victor Miller of IBM and Neal Koblitz from the University of Washington [1, 2]. The security of ECC is based on the hardness of elliptic curve discrete logarithm problem (ECDLP). ECC can improve the performance of SSL because ECC has smaller key length but provides the same security level compared with RSA. Smaller key length results in faster computation, lower power consumption, and lower memory and bandwidth. Table 1 shows the equivalent key sizes of ECC and RSA [6]. Currently, 1024-bit RSA is standard, but it is projected that the size will increase to 2048 bits after 2010. The performance issues of RSA with such a large key size will then become a dominant force, which can severely affect the performance of RSA. Therefore, we shall use 283-bit ECC in place of the 2048-bit RSA since it can significantly reduce the key length but still provides the same security level.

Table 1: Equivalent Key Sizes between ECC and RSA.

ECC	RSA	Protection lifetime
163	1024	until 2010
283	3072	until 2030
409	7680	beyond 2031

Despite ECC's advantages over RSA, software based ECC implementations usually require long computation time, hence makes it difficult to be effectively utilized in real-time web-based transactions. To overcome this drawback, we propose an efficient FPGA implementation of ECC over GF(2^{283}), where GF stands for Galois Field, and 2^{283} means 283-bit binary operation. The key arithmetic operation in ECC is point multiplication. It determines the performance of the elliptic curve cryptosystem because it is the most computationally expensive unit. The main contribution of our FPGA based design is the resources sharing and parallel processing optimization. The simulation results show that our implementation is significantly faster than the software implementation as well as previous FPGA implementations with the same security level [9, 10].

The rest of this paper is organized as follows. In Section 2, we review previous hardware implementations of ECC. In Section 3, we provide the algorithms of ECC. In Section 4, we present the detailed design of point multiplication. In Section 5, we show the experimental results. In Section 6, we conclude our work.

2 Previous Work

Hardware implementation of ECC has better performance than software implementation. Existing hardware implementations vary in the following aspects: GF(2^m), GF(p), key lengths (from 163 bits to 233 bits), platforms (FPGA, ASIC, sensor). In this section, we review some of the FPGA implementations of ECC over GF(2^m).

Orlando and Paar designed a reconfigurable elliptic curve processor (ECP) over GF(2^{167}) [7]. The ECP consists of main controller, arithmetic unit controller and arithmetic units. The point multiplication can be computed in 0.21 ms using the Montgomery algorithm. This work is generally considered as the benchmark of FPGA implementation of ECC. Its main advantages include scalable hardware architecture and reprogrammable processing units. Sandoval and Uribe proposed a hardware architecture that can perform three different ECC algorithms, i.e. elliptic curve Diffie-Hellman (ECDH), elliptic curve digital signature (ECDSA), elliptic curve integrated encryption scheme (ECIES) [8]. The main functional units in their cryptosystem are: coprocessor for scalar multiplication, random number generator, algorithms modules, and main controller. Its scalar multiplication can be completed in 4.7 ms for GF(2^{191}). Ernst et al. presented a generator based elliptic curve cryptosystem in [9]. The generator program can create customized VHDL netlists according to different key sizes and multiplier radix. Thus, this work is flexible in validating the correctness of the design. The authors chose Massey-Omura finite field multiplier, and Double-and-add algorithm for point multiplication. Their point multiplication can be computed in 6.85 ms for GF(2^{270}). Later, Leung et al. presented a microcoded FPGA based elliptic curve processor [10], which is similar to that presented in [9]. This design is parameterized for arbitrary key sizes and it allows for rapid development of different control flows. They used normal basis for the Galois field operations and the point multiplication can be computed in 14.3 ms for GF(2^{281}).

In addition to the hardware implementations discussed above, there exist other FPGA implementations for binary field in literature, such as [11, 12, 13]. A survey study conducted by Dormale and Quisquater is presented in [14], which summaries these FPGA based implementations.

3 Review of Elliptic Curve Cryptography

We first review the design hierarchy of a typical elliptic curve cryptosystem. Then, we describe the algorithms to compute the point multiplication.

3.1 Design Hierarchy

The design hierarchy of a typical elliptic curve cryptosystem is shown in Fig. 1. The top level of the system contains cryptographic protocols. In an ECC based SSL connection, the ECC based cipher suite uses ECDH for key exchange, and ECDSA for authentication of the public key. Point multiplication is utilized in both of the ECDH and ECDSA protocol. The secondary level in the design hierarchy is point multiplication. Point multiplication is composed of point doubling and point addition. Point multiplication, point doubling and point addition are operations involving with the points on the elliptic curve. The bottom level of the ECC system is Galois field arithmetic including Galois field multiplication, Galois field inversion and Galois field squaring. Our design focuses on all but the protocol level of the elliptic curve cryptosystem.

3.2 Algorithms of ECC

According to the group law of points on elliptic curve E, both point addition and point doubling need a Galois field inversion [3]. Galois field inversion is much more expensive than Galois field multiplication. Using projective coordinates can eliminate the use of Galois field inversion in point addition and point doubling. The point addition and point doubling in projective coordinates can be computed as following [15]:

- Point addition in projective coordinates:

$$Z_3 = (X_1 \cdot Z_2 + X_2 \cdot Z_1)^2 \tag{1}$$

$$X_3 = x \cdot Z_3 + (X_1 \cdot Z_2) \cdot (X_2 \cdot Z_1) \tag{2}$$

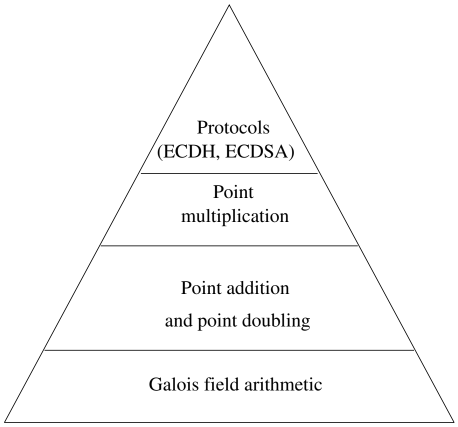

Figure 1: Hierarchy of Typical Elliptic Curve Cryptosystem.

where (X_3, Z_3) is the result of the point addition in projective coordinate, and (X_1, Z_1) (X_2, Z_2) are the projective coordinates of P and Q, respectively.

- Point doubling in projective coordinates:

$$Z = X_1^4 + b \cdot Z_1^4 \tag{3}$$

$$X = Z_1^2 \cdot X_1^2 \tag{4}$$

where (X, Z) is the result of the point doubling in projective coordinates, and (X_1, Z_1) is the projective coordinates of P.

In our design, we use Montgomery point multiplication algorithm for the implementation of point multiplication [3, 12, 15]. The pseudocode is shown in Algorithm 1.

Algorithm 1 Montgomery Point Multiplication Algorithm.

INPUT: An integer $k = (k_{n-1}, k_{n-2}, \cdots k_1, k_0$, $k_{n-1} = 1$, a point $P(x, y) \in E(GF(2^m))$
OUTPUT: $Q = kP$.
Set $X_1 = x, Z_1 = 1, X_2 = x^4 + b, Z_2 = x^2$
for $i = n - 2$ downto 0 **do**
  if $k_i = 1$ **then**
    Pointadder(X_1, Z_1, X_2, Z_2),
    Pointdouble(X_2, Z_2)
  else
    Pointadder(X_2, Z_2, X_1, Z_1),
    Pointdouble(X_1, Z_1)
  end if
end for
return $Q = M_{xy}(X_1, Z_1, X_2, Z_2)$.

Note, "Pointadder" and "Pointdouble" in Algorithm 1 are computed using Equations (1) - (4). M_{xy} is the function to convert the projective coordinates to affine coordinates [11]. Its output, i.e., the coordinate of point Q, x_k and y_k can be computed as:

$$x_k = \frac{X_1}{Z_1} \tag{5}$$

$$y_k = (x+x_k)[(y+x^2)+(\frac{X_2}{Z_2}+x)(\frac{X_1}{Z_1}+x)]\times\frac{1}{x}+y \tag{6}$$

4 Architecture and Implementation of Point Multiplier

The top level architecture of a typical elliptic curve cryptosystem is illustrated in Figure 2. It is composed of main controller, register files, and point multiplier. The main controller is used to realize specific cryptographic protocols, such as ECDSA or ECDH. Point multiplier consists of point adder, point doubler and conversion module. And its implementation is our focus in this work. Details of the implementation of point multiplier is described in the next section.

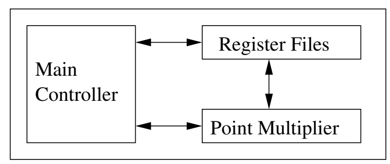

Figure 2: Top Level View of the Elliptic Curve Cryptosystem.

The diagram of the point multiplier is shown in Figure 3. Based on the Montgomery point multiplication algorithm, the point multiplier is composed of point adder, point doubler, coordinates converter, squarer and XORs.

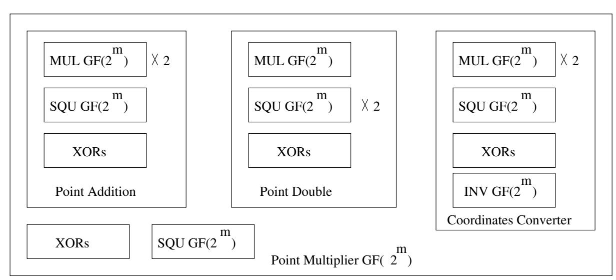

Figure 3: Architecture of Point Multiplier.

We use two Galois field multipliers, one Galois field squarer and XORs to implement point adder. Point doubler is composed of two Galois field squarers, one Galois field multiplier and XORs. The coordinates converter is more complicated than point adder and point doubler. It consists of two Galois field multipliers, one Galois field squarer, one Galois field inverter, and XORs. In our work, all the arithmetic units are designed in GF(2^{283}). The goal of our design is to optimize the parallel processing of the Montgomery point multiplication. Meanwhile, our design shares the arithmetic units in order to reduce the chip area.

4.1 Adder in Galois Field

The addition unit in Galois field is straightforward to implement over binary field. It can be designed using an array of XOR gates.

4.2 Squarer in GF(2^{283})

We have designed a bit parallel squarer which runs much faster than simply multiplying two binary polynomials. Assume the binary polynomial is $a(x) = \sum_{i=0}^{282} a_i x^i$, then $a(x)^2$ can be calculated by Equation (7) [3]:

$$a(x)^2 = \sum_{i=0}^{282} a_i x^{2i} \ modulo \ f(x) \tag{7}$$

We choose $f(x) = x^{283} + x^{12} + x^7 + x^5 + 1$ as the reduction polynomial recommended by ANSI [3]. So we can derive the formula to compute the coefficients of $a(x)^2$ by replacing x^n with $(x^{12}+x^7+x^5+1)\cdot x^{n-283}$, where $n \geq 283$. Therefore, the squarer is simply a set of XOR arrays to recombine the coefficients of $a(x)$. And the gate count is proportional to the polynomial bit [19], which is 283 in our case.

4.3 Multiplier in GF(2^{283})

To implement the multiplier, we adopt the digit serial multiplier introduced in [3]. It has the advantage of being able to increase the speed of the multiplication operation. The digit serial multiplier needs to use a reduction module. The algorithm to design the digit serial multiplier is shown in Algorithm 2.

Algorithm 2 Digit Serial Multiplier in GF(2^m).

INPUT: $a = \sum_{i=0}^{m-1} a_i z^i \in$ GF(2^m), $b = \sum_{i=0}^{l-1} B_i z^{ki} \in$ GF(2^m), reduction polynomial $f(z)$
OUTPUT: $c = a \cdot b$
Set $c = 0$
for $i = 0$ to $l-1$ **do**
  $c = c + B_i a$
  $a = a \cdot z^k \bmod f(z)$
end for
return $c \bmod f(z)$

In Algorithm 2, $l = \lceil m/k \rceil$, k is the digit size, l is the number of digits. In our implementation, we set $m = 283$, $k = 32$, $l = 9$. Using this digit serial multiplier can improve the performance of the Galois field multiplication compared to bit serial multiplier.

4.4 Reduction in GF(2^{283})

The reduction function is used in designing the multiplier. We choose the fast reduction modulo algorithm with digit size of 32 in our implementation [3]. The pseudocode is shown in Algorithm 3.

Algorithm 3 Fast Reduction Modulo in GF(2^{283}).

INPUT: A binary polynomial $c(z)$ of degree at most 564
OUTPUT: $c(z) \bmod f(z)$
for $i = 17$ downto 9 **do**
  $T = C[i]$
  $C[i-9] = c[i-9] + (T << 5) + (T << 10) + (T << 12) + (T << 17)$
  $C[i-8] = c[i-8] + (T >> 27) + (T >> 10) + (T >> 12) + (T >> 17)$
end for
$T = C[8] >> 27$
$C[0] = C[0] + T + (T << 5) + (T << 7) + (T << 12)$
$C[8] = C[8] \,\&\, 0x7FFFFFF$
return $(C[8], C[7], \cdots C[1], C[0])$

Note, $C[i]$ is a 32-bit word of $c(z)$, i.e. $c(z) = (C[17], C[16], \cdots C[0])$, which is at most 564-bit long. And the reduction result only consists of $(C[8], C[7], \cdots C[0])$, which has bit width of 283. The reduction module is composed of shift registers, XORs, and AND gates. One notable feature of our reduction module is that it can finish the computation in 4 clock cycles.

4.5 Inverter in GF(2^{283})

Inversion is the most complex operation in Galois field arithmetic. It is based on Fermat's little theorem [16]. Let α be a nonzero element in GF(2^{283}), then $\alpha^{-1} = \alpha^{2^{283}-2}$. We can see that $2^{283} - 2 = \sum_{i=1}^{282}$. Thus,

$$\alpha^{-1} = \alpha^{\sum_{i=1}^{282} 2^i} = \prod_{i=1}^{282} \alpha^{2^i} \tag{8}$$

According to equation (8), the inversion can be implemented using 282 squarings and 281 multiplications. Actually, the number of multiplications can be reduced because of the following features [3, 18].

- When m is odd:

$$\alpha^{2^{m-1}-1} = (\alpha^{2^{\frac{m-1}{2}}-1})^{2^{\frac{m-1}{2}}} \alpha^{2^{\frac{m-1}{2}}-1}$$

- When m is even: $\alpha^{2^{m-1}-1} = (\alpha^{2^{m-2}-1})^2\alpha$

Based on the above, we derive the following formula to compute the inversion in GF(2^{283}).

$$\begin{array}{rclcl} tmp1 & = & \alpha^{2^2-1} & = & \alpha \cdot \alpha^2 \\ tmp2 & = & \alpha^{2^4-1} & = & tmp1 \cdot (tmp1)^{2^2} \\ tmp3 & = & \alpha^{2^8-1} & = & tmp2 \cdot (tmp2)^{2^4} \\ tmp4 & = & \alpha^{2^{16}-1} & = & tmp3 \cdot (tmp3)^{2^8} \\ tmp5 & = & \alpha^{2^{17}-1} & = & \alpha \cdot (tmp4)^2 \\ tmp6 & = & \alpha^{2^{34}-1} & = & tmp5 \cdot (tmp5)^{2^{17}} \\ tmp7 & = & \alpha^{2^{35}-1} & = & \alpha \cdot (tmp6)^2 \\ tmp8 & = & \alpha^{2^{70}-1} & = & tmp7 \cdot (tmp7)^{2^{35}} \\ tmp9 & = & \alpha^{2^{140}-1} & = & tmp8 \cdot (tmp8)^{2^{70}} \\ tmp10 & = & \alpha^{2^{141}-1} & = & \alpha \cdot (tmp9)^2 \\ tmp11 & = & \alpha^{2^{282}-1} & = & tmp10 \cdot (tmp10)^{2^{141}} \\ \alpha^{-1} & = & \alpha^{2^{283}-2} & = & (tmp11)^2 \end{array} \quad (9)$$

where $tmp1$ to $tmp11$ are 283-bit registers used to store temporary data for the inversion operations. According to Equation (9), we can figure out that the inversion only needs 11 multiplications and 282 squarings, which is quite efficient.

5 Experimental Results and Analysis

We have implemented and simulated the elliptic curve point multiplication with Xilinx's FPGA device. In order to show the effectiveness of hardware implementation over software based approach, we have also realized the design in software. We first provide the setups used in our work, then compare our FPGA based design with several previous works, and then show the difference between hardware and software implementations.

5.1 Software Implementation

The software implementation of the elliptic curve point multiplication is done using C++ and LiDIA. LiDIA is a C++ library of computational number theory [17]. The simulation of the point multiplication in GF(2^{283}) is based on Algorithm 1 and carried out on a Pentium4 2.8 GHz desktop with 1G memory. The source codes are compiled by GCC 4.1.1. The running time to perform a single Tate pairing operation is 9.6 ms.

5.2 FPGA Implementation

The hardware implementation is simulated by ModelSim XE and synthesized with Xilinx ISE 8.2i. The target device is Xilinx Virtex 4 XC4VFX140-FF1517-11. The optimization goal during synthesis is set as "speed", and the optimization effort is set to "normal".

We have simulated the elliptic curve point addition, point doubling, coordinates converter and point multiplication in both software and hardware. The simulated latencies for these operations are shown in Table 2. Here, latency is the time to perform one specific arithmetic operation. The k values in our simulation have the same number of 1's and 0's in the binary representation. Point multiplication is the slowest module among other modules because it is composed of point addition, point doubling and coordinates converter.

According to Table 2, the FPGA implementation of the point multiplication is 31.6 times faster than the software implementation. We compare the simulated latency with Leung's [10] and Ernst's work [9] and show the results in Table 3. Our FPGA implementation of the point multiplication is 47 times faster than that in Leung's work (13.3 ms), and 22.5 times faster than that in Ernst's work (6.85 ms).

Table 3: Comparisons of Latency of Point Multiplication.

Design	Key Size	Latency
[10]	281	14.3 ms
[9]	270	6.85 ms
Our work	283	0.304 ms

6 Conclusion

In this paper, we study hardware implementation of elliptic curve point multiplication to speedup secure web transactions. We propose an FPGA based implementation in GF(2^{283}) optimizing the parallel processing and resources sharing. Our implementation is significantly faster than those previous works presented in the literature. We also compared the FPGA implementation with its software implementation. The experimental results show that the hardware based implementation can improve the latency by a factor of 31.

References

[1] V. S. Miller, Use of Elliptic Curves in Cryptography, *Advances in Cryptology*, Vol. 218, pp. 417-426,

Table 2: Speedup of Hardware over Software.

Operation	FPGA Freq. (MHz)	FPGA Latency (us)	Software Latency (us)	Speedup
Point Addition GF(2^{283})	283.728	0.6	29	48
Point Doubling GF(2^{283})	281.861	0.41	21	51
Coordinates Converter	183.968	24	58	2.4
Point Mult. GF(2^{283})	171.247	304	9600	31.6

1985.

[2] N. Koblitz, Elliptic Curve Cryptosystems, *Mathematics of Computation*, Vol. 48, pp. 203-209, 1987.

[3] Darrel Hankerson, Alfred Menezes, and Scott Vanstone, *Guide to elliptic curve cryptography*, Springer, 2004.

[4] R. Rivest, A. Shamir, and L. Adleman, A Method for obtaining digital signatures and public-key cryptosystems, *Communications of the ACM*, Vol. 21, pp. 120-126, 1978.

[5] C. Coarfa, P. Druschel, and D. Wallach, Performance Analysis of TLS Web Servers, *Network and Distributed Systems Security Symposium*, 2002.

[6] V. Gupta, S. Gupta, and S. Chang and D. Stebila, Performance analysis of elliptic curve cryptography for SSL, *3rd ACM workshop on wireless security*, pp. 87-94, 2002.

[7] Gerardo Orlando and Christof Paar, A High-Performance Reconfigurable Elliptic Curve Processor for GF(2^m), *Second International Workshop on Cryptographic Hardware and Embedded Systems - CHES 2000*, pp. 41-56, 2000.

[8] M. Morales-Sandoval and C. Feregrino-Uribe, On the hardware design of an elliptic curve cryptosystem, *Proceedings of the Fifth Mexican International Conference in Computer Science*, pp. 60-70, 2004.

[9] M. Ernst, S. Klupsch, O. Hauck, and S.A. Huss, Rapid Prototyping for Hardware Accelerated Elliptic Curve Public-Key Cryptosystems, *12th International Workshop on Rapid System Prototyping*, pp. 24-29, 2001.

[10] Leung, K.H., Ma, K.W., Wong, W.K., and Leong P.H.W., FPGA implementation of a microcoded elliptic curve cryptographic processor, *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 68-76, 2000.

[11] B. Ansari and M. Anwar Hasan, High performance architecture of elliptic curve scalar multiplicatioin, *Tech. Report CACR*, 2006.

[12] Nazar A. Saqib, Francisco Rodriguez-Henriquez, and Arturo Diaz-Perez, A Parallel Architecture for Fast Computation of Elliptic Curve Scalar Multiplication over GF(2^m), *18th International Parallel and Distributed Processing Symposium*, pp. 26-30, 2004.

[13] C. Shu, K. Gaj, and T. El-Ghazawi, Low latency elliptic curve cryptography accelerators for NIST curves on binary fields, *IEEE Field-Programmable Technology (FPT)*, pp. 309-310, 2005.

pp. 303-306, 2004.

[14] Guerric Meurice de Dormale and Jean-Jacques Quisquater, High-speed hardware implementations of Elliptic Curve Cryptography: A survey, *J. Syst. Archit.*, Vol. 53, pp. 72-84, 2007.

[15] Julio Lopez and Ricardo Dahab, Fast Multiplication on Elliptic Curves over GF(2^m) without Precomputation, *Cryptographic Hardware and Embedded Systems*, pp. 316-327, 1999.

[16] J. Guajardo and C. Paar, Itoh-Tsujii Inversion in Standard Basis and Its Application in Cryptography and Codes, *Designs, Codes and Cryptography*, Vol. 25, pp. 207-219, 2002.

[17] LiDIA - A Library for Computational Number Theory, http://www.cdc.informatik.tu-darmstadt.de/TI/LiDIA/.

[18] Toshiya Itoh and Shigeo Tsujii, A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases, *Information and Computation*, vol. 78, pp. 171-177, 1988.

[19] Chang Shu, Soonhak Kwon, and Kris Gaj, FPGA Accelerated Tate Pairing Based Cryptosystems over Binary Fields, *IEEE International Conference on Field Programmable Technology*, 2006.