New Generation Cryptosystems using Elliptic Curve Cryptography

Sharat Narayan

Abstract— **Elliptic Curve Cyrptography is emerging as an attractive public key cryptosystem for mobile/wireless environment. This is mostly due to the small key sizes associated with Elliptic Curve Cryptograhic systems. This project presents the different uses of Elliptic curve cryptography in New Generation Cryptosystems - the internet and in mobile computing. Different hardware and software implementations suited for this purpose are discussed. The OpenSSL implementation of ECC is also discussed in this project.**

INTRODUCTION

Since its proposal by Victor Miller and Neal Kolbitz in the mid 1980s, Elliptic Curve Cryptography (ECC) has evolved into a mature public-key cryptosystem. Extensive research has been done on the underlying math, its security strength and efficient implementations.

Small key sizes and computational efficiency of both public and private key operations make ECC not only applicable for hosts running on powerful machines but also to small wireless devices such as PDA's and cell phones. To make ECC commercially viable, ECC protocols need to be standardized and integrated seamlessly into the security layers like SSL.

This paper is structured as follows: Section 1 gives an overview of ECC and the existing protocols like ECDH and ECDSA. Section 2 discusses the integration of ECC into SSL and the performance improvement gained over other public key cryptosystems is analysed. Section 3 gives the architecture of an ECC harware accelerator and the implemented algorithms.

I. ECC OVERVIEW

Every public key cryptosystem is based on some hard to solve mathematical property. By hard to solve we mean that even on the fastest of computers available today it is infeasible in terms of money and computational time to solve the problem. The popular RSA and Deffie-Hellman are based on the hardness of integer factorization ad Discrete Logarithm Problem. Unlike these cryptosystems Elliptic Curve Cryptography operates on points on a Elliptic Curve. The basic operation in ECC is the point multiplication i.e. multiplication of an elliptic curve point P by an integer e, which we will denote by $e * P$. It is equivalent to adding P to itself e times, which yields another point on the curve.

Similar to the Discrete Logarithm Problem we have the Elliptic Curve Discrete Lograthm which is finding K for a curve such that $P = k * Q$ given P and Q. This is possible by the brute force approach which is to compute all multiples of Q until P is found. In real cryptographic systems where the key size is large follwoing the brute force approach to solve the Elliptic curve discrete logarithm problem is infeasible.

The Security of the Elliptic curve cryptography relies on the hardness of solving the elliptic curve discrete logarithm problem. Not every elliptic curve offers strong security properties and for some curves the elliptic curve discrete logarithm problem may be solved efficiently. Since poor choice of the curve can compromise security, standard organizations like NIST and SECG have published a set of recommended curves.

The Elliptic Curve Diffie-Hellman (ECDH) and Elliptic Curve Digital Signature Algorithm (ECDSA) are the elliptic curve counterparts Diffie-Hellman and DSA. In ECC we have one more important aspect, the base point P. This base point P is fixed for each curve. In ECC this base point is used to calculate the public key. A random integer k is chosen and is kept private and forms the secret key. The result of the multiplication $Q = k * P$ forms the public key of the cryptosystem.

A. Elliptic Curve Diffie-Hellman (ECDH)

This protocol establishes a shared key between two parties. The original Diffie-Hellman algorithm is based on multiplicative group modulo p, while the ECDH protocol is based on the additive elliptic curve group. We assume that the underlying field $GF(p)$ or $GF(2^k)$ is selected and the curve E with parameters a, b and the base point P is set up. The order of the base point P is equal to n. The standards often suggest that we select an elliptic curve with prime order, and therefore, any element of the group would be selected and their order will be the prime number n. At the end of the protocol the communicating parties end up with the same value K which is point on the curve. A part of this value can be used as a secret key to a secret-key encryption algorithm.

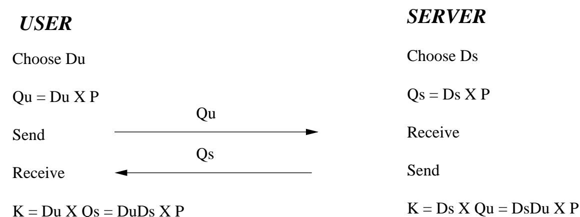

Fig. 1. ECDH

B. Elliptic Curve Digital Signature Algorithm (ECDSA)

First, an elliptic curve E defined over $GF(p)$ or $GF(2^k)$ with large group of order n and a point P of large order is selected and made public to all users. Then, the following key

generation primitive is used by each party to generate the individual public and private key pairs. Furthermore, for each transaction the signature and verifiction primitives are used. ECDSA is briefly outlined below,

ECDSA Key Generation - The user A follows these steps:

1. Select a random integer $d\epsilon[2, n-2]$.
2. Compute $Q = d * P$.
3. The public and private keys are (E, P, n, Q) and d.

ECDSA Signature Generation - The user A signs the message m using these steps

1. Select a random integer $k\epsilon[2, n-2]$.
2. Compute $k * P = (x1, y1) andr = x1 modn$.
   if $x1\epsilon GF(2^k)$, $x1$ is represented as a binary number.
   if $r = 0$ then go to Step 1.
3. Compute $k^{-1} modn$
4. Compute $s = k^{-1}(H(m) + dr) modn$.
   Here H is the secure hash algorithm SHA
   If s = 0 go to step 1.
5. The Signature for the message m is (r, s)

ECDSA Signature Verification-The User B Verfies A's Signature (r, s) on the message m by applying the following steps:

1. Compute $c = s^{-1} modn$ and H(m)
2. Compute $u1 = H(m) cmodn$ and $u2 = rcmodn$
3. Compute $u1 * P + u2 * Q = (x0, y0)$ and $v = x0 modn$.
4. Accept the signature if $v = r$

II. SSL Operation

A. Overview

Secure Socket Layer (SSL) is the most widely used and deployed security protocol on the internet today. Currently it is supported as HTTPS. It is now trusted to secure virtually all sensitive web-based applications ranging from online banking and stock trading to e-commerce.

SSL offers encryption, source authentication and integrity of data that is transmitted over an insecure channel. It operates over a reliable protocol like TCP. It is very flexible in the sense that it can accomodate diffrent cryptographic algorithms for key agreement, encryption and hashing. However, the specifications does recommend particular combinations of these algorithms called $cipher - suites$. For example a cipher-suite such as RSA-RC4-MD5 would indicate that RSA will be used for key exchange mechanism. RC4 for bulk-encryption and MD5 for hashing.

There are 2 main components in a SSL, the Handshake protocol and the Record Layer Protocol. The Handshake protocol is used by the client and the server to agree on a common cipher suite, authenticate each other and establish a shared master key using public key cryptographic algorithms. The Record Layer Protocol then derives symmetric keys from the master key which are then used for bulk encryption and authentication of source data.

Public-key cryptographic operations are the most computationally expensive portion of SSL processing. SSL allows the re-use of a previoulsy established master secret resulting in an abbreviated handshake that does not involve any public-key cryptography, and requires fewer and shorter messages. However, a client and server must perform a full handshake on their first interaction.

B. RSA Based Handshake

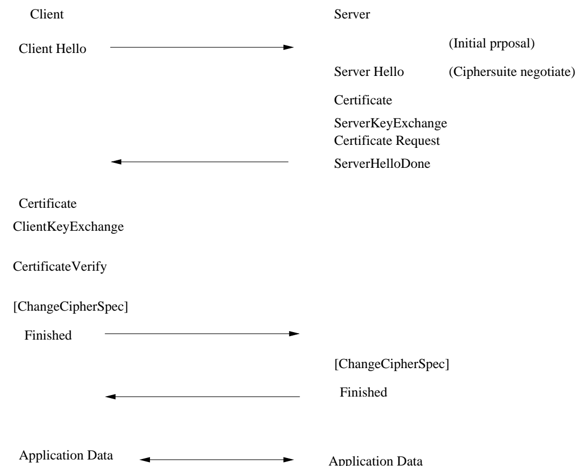

Fig. 2. RSA based SSL Handshake

As of today the mostly commonly used public-key cryptosystem for key establishment is RSA. Figure 1 shows the operation of an RSA-based SSL handshake.

The client and the server exchange random nonces initially and negotiate a cipher suite with $ClientHello$ and $ServerHello$ messages. The server then sends its signed RSA public key either in the $ServerCertificate$ message or the $ServerKeyExchange$ message. The client after verifying the public key generates a random 48 - byte number, encrypts it with the server's public key, and sends it in the $ClientKeyExchange$ message. The server decrypts to message to get the pre-master key. Both the end-points then use premaster secret which, along with previously exchanged nonces, is used to derive the symmetric keys. The server has the option of doing client authentication.

In a RSA Handshake without client authentication, the client performs two RSA public-key operations - one to verify the server's certificate and another to encrypt the premaster secret with the server's public key. The server does one public key operation to decrypt the premaster secret key sent by the client.

C. ECC Based Handshake

The ECC based handshake is very similar to RSA handshake. Only the handshake protocol is affected by incorporating ECC in SSL. The same messages as shown in figure 2 are exchanged between the client and the server. Only the third message, $ServerKeyExchange$ is absent in the ECC based handshake.

Through the first 2 message the client and server agree on the ECC based cipher-suite. In this case the $SeverCertificate$ contains the ECDH public key signed by a certificate authority using ECDSA. The client after validating the ECDSA signature conveys its ECDH public key to server in the $ClientKeyExchange$ message. The two entities now perform the ECDH opertaions using its own private key and others' public key. At the end of the ECDH operation they get the master

secret and symmetric key. Client authentication is optional in this protocol also.

In a ECC handshake without client authentication, the client performs ECDSA verification to verify the server's certificate. It also does one ECDH operation using its private key and server's public key to obtain the premaster key. The server needs to perform only ECDH operation to arrive at the same secret.

D. Performance comparison between ECC based Handshake and RSA based handshake

The RSA Based Handshake is the most widely used public-key cryptosystem in SSL and has been in use for quite some time. The ECC based handshake was introduced very recently into the SSL. This was mainly made possible by the cryptographic research group at Sun Microsystems Ltd. Viupl Gupta and et al [5]. Their paper presents an estimate of performance improvements that can be expected in SSL through the use of ECC.

The research group at Sun Microsystems conducted their experiment on an Ultra-80 platform(A Sun server equipped with 450 MHz UltraSparc II processor) and Yopy (A Linux PDA equipped with 200 MHz StrongARM processor).

The following cases were considered to compare RSA (1024-bits) and ECDH-ECDSA (163-bits) handshakes.

Case 1: A Yopy to another Yopy

Case 2: A Yopy client talking to an Ultra80 server

Case 3: An Ultra80 talking to another Ultra80

Given below are the results obtained from the experiment.

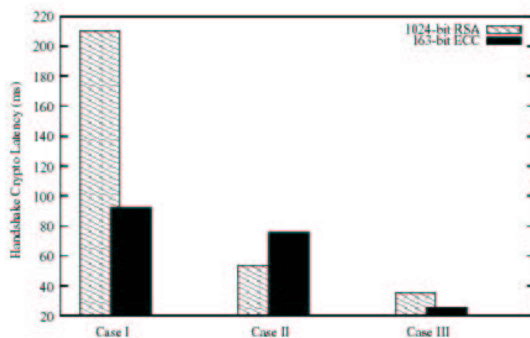

Fig. 3. Case I , CaseII and CaseIII - Without Client Authentication

From the graphs we observe that in case 2 and case 3, ECC handshake protocol has significantly low latency when compared to RSA handshake, incidently this happens when both the client and the server are run on the same platform. When the client and the server run on different platforms RSA handshake performs better. To verify the earlier results these experiments were repeated using higher key sizes. 2048-bit RSA was compared with 193-bit ECC. The ECC was found to perform as expected, even in Case 2. The results are shown in the graph below.

These results indicate that the performance advantage of ECC over RSA increases at higher key sizes.

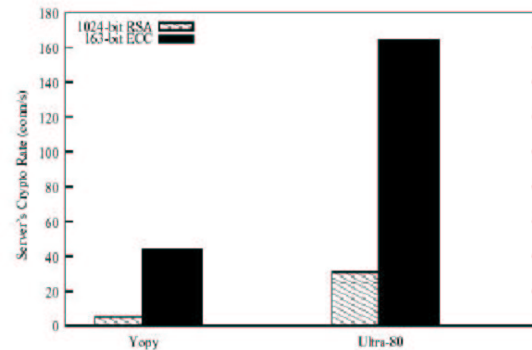

Fig. 4. Case II with higher key size - Without Client Authentication

III. ECC Hardware Acceleration

Until this point we have discussed the different ECC (ECDSA, ECDH) operations in existence and their integration into the widely publicized security layer, the SSL. We have also seen the performance enhancements that can be achieved when ECC is used. The performance efficiency of ECC enables it to be incorporated into clients that range from mobile devices like PDA and cell phones to high end clients like PC's. The aggregation of clients initiated connection leads to high computational demand on the server side, which is best handled by hardware solution. While the client needs are limited as they can be optimized for a particular curve, the server-side hardware needs to be able to operate on numerous curves. The reason for this is that clients may choose different key sizes and curves depending on its preference and the server is expected to support most of the clients.

In order to accelerate the ECC related computations on a web server, Sun Microsystems came up with a cryptographic hardware accelerator for elliptic curve. The complete ECC-enabled Hardware/Software stack is shown below.

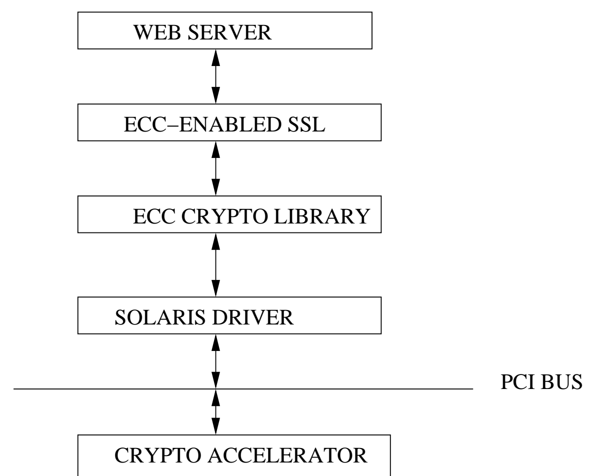

Fig. 5. ECC Hardware/Software stack

A. Architecture of the accelerator

The accelerator was designed such that it could do finite field arithmetic for $GF(2^n), n <= 255$ on arbitrary irreducible

polynomials. It had microprogrammable architecture where the instructions could be overlapped there by enabling parallel instruction execution. It had bus-based data path. The design of the hardware was driven by the need to both provide high performance for named elliptic curves and support point multiplications for arbitrary, less frequently used curves.

The block diagram of the accelerator is as shown in figure 6.

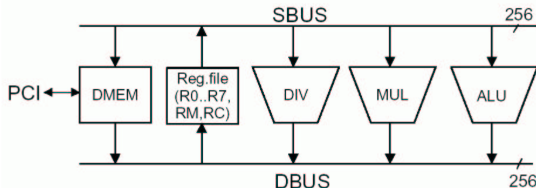

Fig. 6. Accelerator Architecture

The functional units include a modular divider (DIV), a modular multiplier (MUL) and a multifunctional arithmetic and logic uint (ALU). The ALU provides addition, modular squaring, shift and comparison functions.

1) Instruction Set: The ECC processor implements a load-store architecture, that is, only load and store instructions can access memory. All other operations use register. The instructions fall into three categories: Memory instructions, arithmetic instructions and control instructions. All instructions are of fixed length, 16 bits.

2) Control Unit: The control unit consists of the instruction memory IMEM that has a capacity of 1kByte or 512 instructions and a finite state machine that controls the data path according to the instructions fetched.

Program execution times are further optimized by overlapping instruction execution and executing instructions in parallel. In case of a data dependencies, the assembler detects them and prompts the programmer to remove the dependencies. The assembler will not correct these dependencies and considers them as errors. Some dependencies that cannot be eliminated are handled by using NOP instruction.

Parallel execution of instructions is implemented in that an ADD and SQR instruction can be executed in parallel to MUL instruction if there are no data dependencies.

3) Multiplier Unit: The multiplier unit is a digital serial modular multiplier. The design is based on the algorithms described by Song and Parhi. The design is also based on most significant digit multiplier. It was found that by using most significant digit multiplier design pipelining can be done more efficiently.

The multiplier was optimized to do hard-wired reduction for $GF(2^{163}), GF(2^{193})$ and $GF(2^{233})$.

B. Point Multiplication Algorithm

For point multiplication the algorithm used is the Montgomery Scalar Multiplication using projective coordinate as proposed by Lopez and Dahab. This choice of algorithm was motivated by the fact that the processor can perform multiplications faster than divisions.

The Montgomery Scalar Multiplication is done as a projective triple.Montgomerry's algorithm exploits the fact that for a fixed point $P = (X, Y, 1)$ and points $P1 = (X1, Y1, Z1)$ and $P2 = (X2, Y2, Z2)$ the sum $P1 + P2$ can be expressed through only the X and Z co-ordinates of P, $P1$ and $P2$.

The assembly code for Point multiplication is shown below [1]:

```
#Register and values:R0 -> X1, R1 -> Z1,
     R2 -> X2, R3 -> Z3
MUL(R1,R2,R2)
SQR(R1,R1)
MUL(R0,R3,R4)
SQR(R0,R0)
ADD(R2,R4,R3)
MUL(R2,R4,R2)
SQR(R1,R4)
MUL(R0,R1,R1)
SQR(R3,R3)
LD(data_mem_b,R5)
MUL(R4,R5,R4)
SQR(R0,R0)
LD(data_mem_Px,R5)
MUL(R3,R5,R5)
ADD(R$,R0,R0)
ADD(R2,R5,R2)
```

The code is created by considering if it is possible to overlap the instructions. It is also seen that there are no data dependencies for any MUL/SQR or MUL/ADD instruction sequences. Hence, all MUL/SQR and MUL/ADD sequences can be executed in parallel.

C. Performance analysis

In order to analyze the performance of the ECC accelerator over software implementations a hardware was specified in Verilog and prototyped in a Xilinx Virtex XCV2000E-FG680-7 FPGA. The prototype runs off the PCI clock at a frequency of 66.4 MHz.The results of the experimente is as shown in figure7 [1].

	Hardware op/s	Software op/s	Speedup
Named Curves			
$GF(2^{163})$	6987	322	21.7
$GF(2^{233})$	4438	223	19.9
Generic Curves (full)			
$GF(2^{163})$	644	322	2.0
$GF(2^{233})$	451	223	2.0
Generic Curves (part.)			
$GF(2^{163})$	1075	50	21.5
$GF(2^{233})$	757	35	21.6

Fig. 7. Hardware and Software performance

It can be seen from the simulation results that the speedup due to hardware support is enormous for named curves. For

the ECC-163 named curve the hardware accelerator offers 21.7 fold improvement. When generic curves are used the speedup is not enormous but still it is possible to achieve approximately a 2-fold speedup.

IV. Conclusion

The project gave an overview of ECC and two network security protocols. These protocols are used to establish a shared secret key between two parties (ECDH) and to sign a document and then verify the signature (ECDSA). We also saw how ECC can be integrated into the SSL and the performance improvement that can be achieved by doing so.

The hardware support to ECC is also very essential as in the case of web servers that support ECC and we saw the architecture of an ECC accelerator. One of the enhancement that can be made to the accelerator is to improve the execution speed of the control instructions. The the contorl flow instructions like BMZ, BEQ, SL, JMP and END consume almost 21 percent of the execution speed [1]. Latency due to these control flow instructions can be reduced by using techniques like Branch prediction. More instruction level parallelism and loop level parallelism can be introduced into the processor by redesigning the execution stages of an instruction.Several techniques like loop unrolling and software pipelining can be used.

Handling data dependencies is one other area that needs to be looked at, currently the assembler signals an error when data dependencies are found. Smart assemblers can be built such that they analyzes the code and eliminate dependencies in the code. The assembler is also responsible for scheduling the instructions in such a manner that maximum instruction level parallelism is achieved. Such an assembler gives flexibility to the programmer and the programmer can try out new algorithms without concerning himself with execution details.

References

[1] HansEberle, Nils Gura, Sheueling Chang-Shantz - Sun Microsytems. “A Cryptographic Processor for Arbitrary Elliptic Curves over $GF(2^m)$,” .
[2] M.Aydos, E. Savas and C.K.Koc. “Implementating Network Security Protocols based on Elliptic Curve Cryptography”
[3] M.Aydos, T Yanik and C.K.Koc, “An High-Speed ECC based Wireless Authentication Protcol on an ARM Processor,”
[4] M.Aydos, B. Sunar and C.K.Koc. “An Elliptic Curve Cryptography based Authentication and key Agreement Protocol for Wireless Communication,”
[5] Vipul Gupta, Sumit Gupta, Sheueling Chang, ”Performance Analysis of Elliptic Curve Cryptography for SSL”.
[6] Vipul Gupta, Sumit Gupta, Hans Eberle,et al. ”An End-to-End Systems Approach to Elliptic Curve Cryptography”.