Implementing Network Security Protocols based on Elliptic Curve Cryptography *†

M. Aydos, E. Savaş, and Ç. K. Koç
Electrical & Computer Engineering
Oregon State University
Corvallis, Oregon 97331, USA
{aydos,savas,koc}@ece.orst.edu

Abstract

Elliptic curve cryptography provides a methodology for obtaining high-speed, efficient, and scalable implementations of network security protocols. In this paper, we describe in detail three protocols based on elliptic curve cryptographic techniques, and the results of our implementation of the elliptic curve cryptography over the Galois field $GF(2^k)$, where k is a composite number.

1 Elliptic Curve Cryptography

Elliptic curve cryptography [9, 5, 8, 6] provides a methodology for obtaining high-speed, efficient, and scalable implementations of network security protocols. The security of these protocols depends on the difficulty of computing *elliptic curve discrete logarithm* in the elliptic curve group. The group operations utilize the arithmetic of *points* which are elements of the set of solutions of an elliptic curve equation defined over a finite field. The arithmetic of elliptic curve operations depend on the arithmetic on the underlying finite field. The standards suggest the use of $GF(p)$ and $GF(2^k)$. Below, we define the nomenclature and then provide a general overview of security protocols based on elliptic curve cryptography.

- Scalar: An element belonging to either one of the fields $GF(p)$ or $GF(2^k)$ is called a scalar. Scalars are named with lowercase letters: r, s, t, etc.
- Scalar Addition: Two or more scalar can be added to obtain another scalar. In the $GF(p)$ case, this is the ordinary integer addition modulo p. When $GF(2^k)$ is used, this is equivalent to polynomial addition modulo an irreducible polynomial of degree k, generating the field $GF(2^k)$. We will denote the scalar addition of r and s giving the result e by $e = r + s$.

*This research was supported in part by Secured Information Technology, Inc.

†*Proceedings of the Fourth Symposium on Computer Networks*, S. Oktuğ, B. Örencik, and E. Harmancı, editors, pages 130–139, Istanbul, Turkey, May 20-21, 1999.

- Scalar Multiplication: Two or more scalar can be multiplied to obtain another scalar. In the $GF(p)$ case, this is the ordinary integer multiplication modulo p. When $GF(2^k)$ is used, this is equivalent to polynomial multiplication modulo an irreducible polynomial of degree k, generating the field $GF(2^k)$. We will denote the scalar multiplication of r and s giving the result e by $e = rs$.

- Scalar Inversion: The multiplicative inverse of an element a in $GF(p)$ or $GF(2^k)$ is denoted as a^{-1} which is the number with the property $aa^{-1} = 1$. It is often computed using the Fermat's method or the extended Euclidean algorithm.

- Point: An ordered pair of scalars satisfying the *elliptic curve equation* is called a point. Capital letters are used to denote these elements: P, Q, etc. We will also denote a point P using its coordinates $P = (x, y)$, where x and y belong to the field. Furthermore, the x and y coordinates of P will be denoted by $P.x$ or $P.y$, respectively.

- Point Addition: There is a method to obtain a third point R on the curve given two points P and Q, using a set of rules. This is called an elliptic curve point addition. We will use the symbol '+' to denote the elliptic curve addition $R = P + Q$. This should not be confused with scalar addition.

- Elliptic Curve Group: The set of the solutions of the elliptic curve equation together with a special point called *point-at-infinity* form an additive group if the point addition operation defined above is taken as the group operation.

- Point Multiplication: The multiplication of an elliptic curve point P by an integer e will be denoted by $e \times P$. It is equivalent to adding P to itself e times, which yields another point on the curve.

In addition to the above elliptic curve cryptographic primitives, we often need a one-way hash (message digest) function which is defined below:

- Message Digest Function: A message digest function compresses a long message into a short value which is usualy 128 or 160 bits long. Two widely used and standardized hash functions are MD5 and SHA. We will denote the message digest of a message M by $H(M)$. The signature functions take $H(M)$ as an input for efficiency reasons. The hash of the concatenation of two messages M_1 and M_2 is denoted as $H(M_1, M_2)$.

2 Elliptic Curve Diffie-Hellman

This protocol establishes a shared key between two parties. The original Diffie-Hellman algorithm is based on the multiplicative group modulo p, while the elliptic curve Diffie-Hellman (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 a point on the curve. A part of this value can be used as a secret key to a secret-key encryption algorithm.

Figure 1: Elliptic Curve Diffie-Hellmann (Version 1)

User		Server
• Choose $d_u \in [2, n-2]$		• Choose $d_s \in [2, n-2]$
• $Q_u = d_u \times P$		• $Q_s = d_s \times P$
• Send	$\xrightarrow{Q_u}$	• Receive
• Send	$\xleftarrow{Q_s}$	• Receive
• $K = d_u \times Q_s = d_u d_s \times P$		• $K = d_s \times Q_u = d_s d_u \times P$

The second version provides a little more flexibility in the sense that the established value can be preselected by the user and sent to the server. The protocol steps can be modified slightly for sending of a secret value from the server to the user.

Figure 2: Elliptic Curve Diffie-Hellmann (Version 2)

User		Server
• Choose $d_u \in [2, n-2]$		• Choose $d_s \in [2, n-2]$
• $e_u = d_u^{-1} \bmod n$		• $e_s = d_s^{-1} \bmod n$
• $Q = d_u \times K$		
• Send	$\xrightarrow{Q}$	• Receive
		• $R = d_s \times Q = d_s d_u \times K$
• Receive	$\xleftarrow{R}$	• Send
• $S = e_u \times R = e_u d_s d_u \times K = d_s \times K$		
• Send	$\xrightarrow{S}$	• Receive
		• $T = e_s \times S = e_s d_s \times K = K$

3 Elliptic Curve Digital Signature Algorithm

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 verification primitives are used. We briefly outline the Elliptic Curve Digital Signature Algorithm (ECDSA) below, details of which can be found in [4].

ECDSA Key Generation The user A follows these steps:

1. Select a random integer $d \in [2, n-2]$.
2. Compute $Q = d \times P$.
3. The public and private keys of the user A are (E, P, n, Q) and d, respectively.

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

1. Select a random integer $k \in [2, n-2]$.
2. Compute $k \times P = (x_1, y_1)$ and $r = x_1 \bmod n$.
   If $x_1 \in GF(2^k)$, it is assumed that x_1 is represented as a binary number.
   If $r = 0$ then go to Step 1.

3. Compute $k^{-1} \bmod n$.
4. Compute $s = k^{-1}(H(m) + dr) \bmod n$.
   Here H is the secure hash algorithm SHA.
   If $s = 0$ go to Step 1.
5. The signature for the message m is the pair of integers (r, s).

ECDSA Signature Verification The user B verifies A's signature (r, s) on the message m by applying the following steps:

1. Compute $c = s^{-1} \bmod n$ and $H(m)$.
2. Compute $u_1 = H(m)c \bmod n$ and $u_2 = rc \bmod n$.
3. Compute $u_1 \times P + u_2 \times Q = (x_0, y_0)$ and $v = x_0 \bmod n$.
4. Accept the signature if $v = r$.

4 A Mutual Authentication Protocol

Below we describe a mutual authentication protocol which was proposed for a wireless communication environment [1], however, it is suitable for other network environment as well. We assume that there is a certificate authority (CA) which creates and distributes certificates to the users and servers on their request. These certificates contain a temporary identity assigned by the CA for the requesting party, the public key of the requesting party, and the expiration date of the certificate. The concatenated binary string is then signed by the CA's private key to obtain the certificate for the requesting party. By using a certificate the identity of a particular party is bound to its public key. The acquisition of the certificate is performed when the users and servers first subscribe to the service.

4.1 Server and User Initializations

In order to receive a certificate, the server sends its public key Q_s together with its user identity through a secure and authenticated channel to the CA. The CA uses its private key to sign the hashed value of the concatenation of the public key, the temporary identity I_s, and the certification expiration date t_s. The CA then sends the signed message through the secure and authenticated channel to the server as shown in Figure 3.

Figure 3: Server Initialization.

Server		Certification Authority
• Choose $d_s \in [2, n-2]$		• Choose $k_s \in [2, n-2]$
• $Q_s = d_s \times P$		• $R_s = k_s \times P$
• Send	$\xrightarrow{Q_s}$	• Receive
		• Choose unique I_s
		• $r_s = R_s.x$
		• $s_s = k_s^{-1}(H(Q_s.x, I_s, t_s) + d_{ca} r_s)$
	$\xleftarrow{Q_{ca}, I_s, r_s, s_s, t_s}$	• Send
• Receive		
• $e_s = H(Q_s.x, I_s, t_s)$		
• Store $Q_s, Q_{ca}, I_s, r_s, s_s, e_s, t_s$		

Establishing a secure channel from the certification authority to the server is a common and accepted assumption in almost all authentication protocols. In practice the CA may use the postage system as the secure channel to distribute the signed messages and temporary identities stored within a smardcard. The signed message is the certificate of the user which is used in future authentication and key generation processes. By repeating the very same process the user acquires its certificate as shown in Figure 4.

Figure 4: User Initialization.

User		Certification Authority
• Choose $d_u \in [2, n-2]$		• Choose $k_u \in [2, n-2]$
• $Q_u = d_u \times P$		• $R_u = k_u \times P$
• Send	$\xrightarrow{Q_u}$	• Receive
		• Choose unique I_u
		• $r_u = R_u.x$
		• $s_u = k_u^{-1}(H(Q_u.x, I_u, t_u) + d_{ca} r_u)$
	$\xleftarrow{Q_{ca}, I_u, r_u, s_u, t_u}$	
• Receive		• Send
• $e_u = H(Q_u.x, I_u, t_u)$		
• Store $Q_u, Q_{ca}, I_u, r_u, s_u, e_u, t_u$		

The certificate consists of a pair of integers which is denoted as (r_s, s_s) for the server and (r_u, s_u) for the user. Here r_u and r_s are the x coordinates of the (distinct) elliptic curve points R_u and R_s, respectively.

4.2 Mutual Authentication Between User and Server

The protocols shown in Figures 3 and 4 are executed off-line. The mutual authentication and key agreement protocols between the user and the server need to be executed in realtime. We give the combined protocol in Figure 5. Here, we use a secret-key encryption algorithm to encrypt the data in the protocol. A conventional stream cipher (RC4 or SEAL) or a block cipher (DES, 3DES, IDEA, RC5) in the cipher-block-chaining mode can be used. The encryption and decryption operations using the key K acting on the plaintext M and the ciphertext C are denoted as $C := E(K, M)$ and $M := D(K, C)$, respectively.

According to the protocol, whenever there is a service request either by the user or by the server, there is an immediate key exchange. The initiated party will also send a random challenge to the initiating party. Sending the public keys unprotected over the air does not introduce any threat to the security of the system. Once both sides have the other party's public key, they simultaneously generate a secret key to encrypt the data required to have a mutual authentication. This task is accomplished by multiplying the other party's public key Q_2 with this party's private key d_1.

To protect the certificates from an eavesdropper, it is necessary to send the certificates in encrypted form. For this reason, the protocol uses a secret key cipher to encrypt the certificates using the mutually agreed secret key $Q_k.x$. The server encrypts the concatenation of its certificate $e_s, (r_s, s_s)$, the certificate expiration date t_s, and a random number g_s which will be used to obtain the final mutual key of the communication. The final content should also include the challenge if the server is the initiating party. The certificates are usually sent in clear in almost all the other authentication protocols. In our protocol, we

do not reveal the content of the certificate which may be useful for spoofing attacks. This increases the encryption time only slightly since the certificate is not very long (on the order kilobits) and the encryption algorithms are very fast (on the order of megabits per second).

The encrypted message C_0 is then sent to the user. The user then decrypts C_0 and obtains the certificate of the server, the random number g_s and the challenge g_u which in this case sent by itself. Obtaining the original challenge value back from the server confirms the freshnesses of the message and prevents the reply attacks. The user immediately encrypts the concatenation of its certificate $e_u, (r_u, s_u)$, the certificate expiration date t_u, and the random number g_s. This encrypted data which is denoted as C_1 is sent to the server.

Next, the user checks the validity of the certificate, and if it is invalid, the user aborts the communication. On the other side, the server decrypts C_1 and checks whether g_s and the time certificate are valid. If not, it aborts. This mechanism, specifically the use of g_s, defeats *spoofing* attacks by the user side and also prevents unnecessary computation. Next, the server checks the validity of the certificate and accordingly grants or aborts the service. Note that it may be a good idea to generate multiple g values in advance so that the protocol could save some time. However, storing these multiple random numbers will increase the storage requirement of the protocol, which is undesirable.

Figure 5: Mutual Authentication and Key Agreement.

User		Server
• Receive	$\xleftarrow{Q_s}$	• Send
• Generate a random number g_u		
• Send	$\xrightarrow{Q_u, g_u}$	• Receive
• $Q_k = d_u \times Q_s = (d_u d_s) \times P$		• $Q_k = d_s \times Q_u = (d_s d_u) \times P$
• $Q_k.x$: The mutually agreed key		• $Q_k.x$: The mutuallly agreed key
		• Generate a random number g_s
		• $C_0 = E(Q_k.x, e_s, r_s, s_s, t_s, g_u, g_s)$
• Receive	$\xleftarrow{C_0}$	• Send
• $D(Q_k.x, C_0)$: Is g_u present?		
• $C_1 = E(Q_k.x, (e_u, (r_u, s_u), t_u, g_s))$		
• Send	$\xrightarrow{C_1}$	• Receive
		• $D(Q_k.x, C_1)$
		• If g_s and t_u are valid, then
• $c = {s_s}^{-1}$		• $c = {s_u}^{-1}$
• $u_1 = ce_s$		• $u_1 = ce_u$
• $u_2 = cr_s$		• $u_2 = cr_u$
• $R = u_1 \times P + u_2 \times Q_{ca}$		• $R = u_1 \times P + u_2 \times Q_{ca}$
• $v = R.x$		• $v = R.x$
• If $v \neq r_s$, then abort		• If $v \neq r_u$, then abort
• $k_m = h(Q_k.x, g_s, g_u)_{msb-64}$		• $k_m = h(Q_k.x, g_s, g_u)_{msb-64}$
• k_m: The unique secret key		• k_m: The unique secret key

After the verification procedure has been completed by both sides, the user and the server are now ready to use the channel that has been reserved for their communication.

However, there is one more step to complete our goal to have a full secure protocol: To generate a secret key known by each side to encrypt the conversation. They do already have a secret key $Q_k.x$; however, this key cannot be used since it will be the same during the valid time limits of their certificates. Therefore, we need to add a new key exchange step to agree on a unique key to be used for communication during each session. However, we do not prefer to execute another key agreement process due to previously stated power and memory limitations. Instead, we will use the previously generated random numbers g_u and g_s which are known by both sides to generate a new secret key without using the channel again. Both the server and the user perform a hash operation to obtain the new secret key, which we call k_m. This key now can be used for encrypting the data sent through the channel.

5 Elliptic Curves over Composite Fields

An important category of elliptic curve cryptographic algorithms is defined over the finite field $GF(2^k)$. Elliptic curve cryptographic applications require fast hardware and software implementations of the arithmetic operations in $GF(2^k)$ for large values of k. Recently, there has been a growing interest to develop software methods for implementing $GF(2^k)$ arithmetic operations and elliptic curve cryptographic operations [12, 14]. An area of particular interest is the development of efficient software implementations of the arithmetic and elliptic curve operations in $GF(2^k)$, where k is a composite number as $k = nm$. An implementation method for this case was presented in [14], where the authors propose to use the logarithmic table lookup method for the ground field $GF(2^n)$ operations. The field $GF(2^{nm})$ is then constructed using the polynomial basis, where the elements of $GF(2^{nm})$ are polynomials of degree $m-1$ whose coefficients are from the ground field $GF(2^n)$. The field multiplication is performed by first multiplying the input polynomials, and reducing the resulting polynomial by a degree-m irreducible trinomial.

Here, we propose a similar methodology for implementing the arithmetic operations in $GF(2^{nm})$. Our main difference is that we use an optimal normal basis in $GF(2^m)$ to represent the elements of $GF(2^{nm})$ by taking the ground field as $GF(2^n)$. The resulting field operations, multiplication and squaring are quite efficient, and they do not involve modular reductions. Our implementation results indicate that the arithmetic operations in the proposed method are faster than those of [14]. Detailed algorithms for performing field multiplication and inversion operations are reported in [11]. Here, we give a brief overview of our methodology, and report the timing results for the finite field and elliptic curve operations in the special case of $k = 176$, $n = 16$, and $m = 11$.

It is customary to view the finite field $GF(2^k)$ as a k-dimensional vector space defined over the field $GF(2)$. The field over which the vector space defined is generally called the ground field. If we take the ground field as $GF(2)$, then the elements of the k-dimensional vector space are bit strings of length k, i.e., $A = (a_0, a_1, \ldots, a_{k-2}, a_{k-1})$, where $A \in GF(2^k)$ and the entries $a_i \in GF(2)$. However, if $n > 1$ divides k, then it is also possible to select the ground field as $GF(2^n)$. If we take $nm = k$, then, effectively we are constructing an m-dimensional vector space over $GF(2^n)$. The elements of the ground field are represented as bit-strings of length n; the elements of $GF(2^{nm})$ are represented as

$$A = (A_0, A_1, \ldots, A_{m-2}, A_{m-1}) \ ,$$

where the entries $A_i \in GF(2^n)$. These representations, the nm-dimensional vector space

over $GF(2)$ and the m-dimensional vector space over $GF(2^n)$, are equivalent. However, the latter representation is based on n-bit words, and it is more advantageous for implementation on microprocessors, particularly when n is selected properly. For example, $n = 8$ and $n = 16$ have been suggested in [3, 14].

The methodology in [14] uses the values of n as 8 or 16. A basis for $GF(2^8)$ or $GF(2^{16})$ can be chosen; however, this is not important since the field arithmetic is performed using the logarithmic table lookup method. An element of $GF(2^{nm})$ is represented using a degree-$(m-1)$ polynomial whose coefficients are from the field $GF(2^n)$. The multiplication in $GF(2^{nm})$ is performed by first multiplying the operands to obtain the twice-sized polynomial, and then reducing the resulting polynomial by a degree-m irreducible polynomial. In general, this degree-m irreducible polynomial needs to have its coefficients from the field $GF(2^n)$, however, it is well-known [7] that an irreducible polynomial over $GF(2)$ of degree m remains irreducible over $GF(2^n)$ if and only if $\gcd(n, m) = 1$. Therefore, one can select an irreducible polynomial of degree m with coefficients from $GF(2)$ rather than $GF(2^n)$ if $\gcd(n, m) = 1$. Furthermore, the use of a special irreducible polynomial is suggested in [14]. This special polynomial is a trinomial of the form $x^m + x^t + 1$, where $t \leq \lfloor m/2 \rfloor$. Such special trinomials are easy to find; the paper [14] lists them for a few values of m.

Our methodology is similar to the one proposed in [14] in terms of representing the elements of and performing the arithmetic in $GF(2^{nm})$. We also select n as 8 or 16. However, we represent the elements of $GF(2^{nm})$ using an (optimal) normal basis for the field $GF(2^m)$. In this representation, the elements of $GF(2^{nm})$ are m-dimensional vectors whose entries are in $GF(2^n)$. A degree-m irreducible polynomial is selected such that the root of this polynomial generates an optimal normal basis [10]. Furthermore, if $\gcd(n, m) = 1$, then the selected degree-m irreducible polynomial will have its coefficients from $GF(2)$ rather than $GF(2^n)$. This selection provides a much simpler multiplication method. Since $\gcd(n, m) = 1$ and $n = 8$ or 16, we need to have an odd m. This requires that we use an optimal normal basis of type II in the field $GF(2^m)$. For optimal normal bases of type I, m would be even because $m + 1$ is a prime number.

The normal bases are thought to be inefficient for composite fields in this setting [14]. The advantages of (optimal) normal bases seem to disappear for $n > 1$, particularly for squaring operation. However, it is our conclusion that if the ground field operations are computed fast (e.g., using the table lookup method), then the composite field operations can be performed very efficiently in the normal basis. The mathematical and algorithmic details of our methodology can be found in [11].

6 Implementation Results

We have implemented the addition, multiplication, and inversion operations in $GF(2^{176})$, and also the elliptic curve point doubling, addition, and multiplication operations over $GF(2^{176})$. The programs were written in C++ using Microsoft Visual C++ Version 5.0, and executed on a PC with the 300-MHz Pentium II processor, running Windows NT 4.0. Our timing results are given in the first column of Table 1. The elliptic curve operations are performed using the affine coordinate system in which a point is represented using two field elements $P = (x, y)$. The field parameters $a, b \in GF(2^{176})$ are selected randomly. The point multiplication algorithm uses the canonical recoding binary method in which the signed-digit recoding of the 176-bit randomly chosen integer e is used.

Table 1: The timing results for the field and elliptic curve operations.

Operation	Our Method (300-MHz P-II)	Reproduced [14] (300-MHz P-II)	Original [14] (133-MHz P-I)
Field Multiplication	12 μsec	15 μsec	(62.7+1.8) μsec
Field Squaring	1.5 μsec	2.5 μsec	(5.9+1.8) μsec
Field Inversion	60 μsec	63 μsec	160 μsec
EC Addition	80 μsec	83 μsec	306 μsec
EC Doubling	80 μsec	85 μsec	309 μsec
EC Multiplication	25 msec	30 msec	72 msec

We also include the timing results of [14] for comparison. The original timing results of [14] were obtained on a 133-MHz Pentium. We have re-developed the programs of [14] by investing reasonable efforts to optimize the code. The timing results given in the second column are our reproduction of their method. Since the 300-MHz Pentium II processor is about 2.25 times faster than the 133-MHz Pentium, it seems that our reproduction software of the methods of [14] is about 50 % faster than their implementation. In the third column, we also give their original timings on the 133-MHz Pentium processor.

We have obtained all our timings by actual implementation. The results in Table 1 shows that the proposed methodology is faster than our reproduction of their method [14], and about 50 % faster than their reported timings, taking into account the speed difference of the processors.

7 Conclusions

We gave an overview of elliptic curve cryptography and three network security protocols. These protocols are used to establish a shared secret key between two parties (elliptic curve Diffie-Hellman), to sign a document and then verify the signature (elliptic curve DSA), and to establish mutual authentication between two parties, usually a user and a server. These protocols are based on elliptic curve cryptographic techniques. We also provided the results of our implementation of elliptic curve cryptography over the field $GF(2^k)$, where k is a composite number $k = nm$. This implementation uses table lookup techniques to perform the operations in the ground field $GF(2^n)$, while the large field is constructed using the optimal normal basis. The reported timing results only contain the arithmetic operations and the elliptic curve operations. The timings of an entire protocol can be estimated by using these values, however, we note that we are currently implementing the entire steps of these protocols, and will report these results in the near future.

Finally, we note that certain attacks on elliptic curve cryptosystems over composite fields $GF(2^{nm})$ have recently been developed [2, 13]. These attacks are applicable to elliptic curves whose coefficients are defined in the ground field $GF(2^n)$, providing the speedup factor of $\sqrt{2m}$. Furthermore, these attacks are highly efficient on so-called binary anomalous curves, in which the curve coefficients are restricted to the field $GF(2)$. In our proposed implementation, we do not make any assumption that the curve coefficients would be restricted to the ground field $GF(2^n)$. In fact, the timing results summarized in Table 1 are obtained by selecting the parameters a and b randomly in the large field $GF(2^{176})$.

References

[1] M. Aydos, B. Sunar, and Ç. K. Koç. An elliptic curve cryptography based authentication and key agreement protocol for wireless communication. In *2nd International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications Symposium on Information Theory*, Dallas, Texas, October 30, 1998 1998.

[2] R. Gallant, R. Lambert, and S. Vanstone. Improving the parallelized Pollard lambda search on binary anomalous curves. Draft Article, April 7, 1998.

[3] G. Harper, A. Menezes, and S. Vanstone. Public-key cryptosystems with very small key lengths. In R. A. Rueppel, editor, *Advances in Cryptology — EUROCRYPT 92*, Lecture Notes in Computer Science, No. 658, pages 163–173. New York, NY: Springer-Verlag, 1992.

[4] IEEE P1363. Standard specifications for public-key cryptography. Draft version 7, September 1998.

[5] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48(177):203–209, January 1987.

[6] N. Koblitz. *A Course in Number Theory and Cryptography*. New York, NY: Springer-Verlag, Second edition, 1994.

[7] R. Lidl and H. Niederreiter. *Introduction to Finite Fields and Their Applications*. New York, NY: Cambridge University Press, 1994.

[8] A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Boston, MA: Kluwer Academic Publishers, 1993.

[9] V. Miller. Uses of elliptic curves in cryptography. In H. C. Williams, editor, *Advances in Cryptology — CRYPTO 85, Proceedings*, Lecture Notes in Computer Science, No. 218, pages 417–426. New York, NY: Springer-Verlag, 1985.

[10] R. Mullin, I. Onyszchuk, S. Vanstone, and R. Wilson. Optimal normal bases in $GF(p^n)$. *Discrete Applied Mathematics*, 22:149–161, 1988.

[11] E. Savaş and Ç. K. Koç. Efficient composite field arithmetic. Work in progress, May 1999.

[12] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck. Fast key exchange with elliptic curve systems. In D. Coppersmith, editor, *Advances in Cryptology — CRYPTO 95*, Lecture Notes in Computer Science, No. 973, pages 43–56. New York, NY: Springer-Verlag, 1995.

[13] M. J. Wiener and R. J. Zuccherato. Faster attacks on elliptic curve cryptosystems. Draft Article, April 8, 1998.

[14] E. De Win, A. Bosselaers, S. Vandenberghe, P. De Gersem, and J. Vandewalle. A fast software implementation for arithmetic operations in $\mathrm{GF}(2^n)$. In K. Kim and T. Matsumoto, editors, *Advances in Cryptology — ASIACRYPT 96*, Lecture Notes in Computer Science, No. 1163, pages 65–76. New York, NY: Springer-Verlag, 1996.