Workload Characterization of Elliptic Curve Cryptography and other Network Security Algorithms for Constrained Environments

A. Murat Fiskiran and Ruby B. Lee
Princeton Architecture Laboratory for Multimedia and Security (PALMS)
Department of Electrical Engineering
Princeton University
{*fiskiran,rblee*}*@ee.Princeton.edu*

Abstract

In recent years, some cryptographic algorithms have gained popularity due to properties that make them suitable for use in constrained environments like mobile information appliances, where computing resources and power availability are limited. In this paper, we select a set of public-key, symmetric-key and hash algorithms suitable for such environments and study their workload characteristics. In particular, we study elliptic-curve versions of public-key cryptography algorithms, which allow fast software implementations while reducing the key size needed for a desired level of security compared to previous integer-based public-key algorithms. We characterize the operations needed by elliptic-curve analogs of Diffie-Hellman key exchange, ElGamal and the Digital Signature Algorithm for public-key cryptography, for different key sizes and different levels of software optimization. We also include characterizations for the Advanced Encryption Standard (AES) for symmetric-key cryptography, and SHA as a hash algorithm. We show that all these algorithms can be implemented efficiently with a very simple processor.

1. Introduction

With the proliferation of handheld wireless information appliances, the ability to perform security functions with limited computing resources has become increasingly important. In mobile devices such as personal digital assistants (PDAs) and multimedia cell phones [5], the processing resources, memory and power are all very limited, but the need for secure transmission of information may increase due to the vulnerability to attackers of the publicly accessible wireless transmission channel. The problem is further compounded by the fact that security algorithms can be very compute-intensive, which conflicts with the scarce resources available in such a mobile platform.

New smaller and faster security algorithms provide part of the solution. In symmetric-key cryptography for example, the Advanced Encryption Standard (AES) [2,17,18,20] became the new U.S. federal standard for block encryption and it is much leaner and faster than the previous Data Encryption Standard (DES) [19], which it is intended to replace. Likewise, the Elliptic Curve Cryptography (ECC) [1,10,16] provides a faster alternative for public-key cryptography. Much smaller key lengths are required with ECC to provide a desired level of security, which means faster key exchanges, user authentication, signature generation and verification, in addition to smaller key storage needs.

Even though much literature exists about these algorithms that focus on algorithmic optimizations and hardware implementations [6,7,8,23], workload characterization studies are very rare. In this paper, we provide a comprehensive workload characterization of security algorithms suitable for constrained environments. We consider elliptic curve analogs of Diffie-Hellman key exchange (EC-DHKE), ElGamal (EC-ElGamal) and the Digital Signature Algorithm (EC-DSA) for public-key cryptography; the Advanced Encryption Standard (AES) for symmetric-key cryptography; and the Secure Hash Algorithm (SHA) [22] as a hash algorithm for data integrity.

2. Algorithm set

We consider four broad categories of security functions: (1) public-key algorithms, which are required for key exchanges; (2) signature algorithms, which are required for user authentication; (3) symmetric-key algorithms, which are required to encrypt and decrypt messages for confidentiality; (4) hash functions, which are used to verify the integrity of messages. Table 1 shows which specific algorithms are used as representative of each of these classes.

In general, we have chosen to study the newer algorithms from each class that are suitable for constrained environments, and about which there have been few workload characterization studies. The only exception is the Secure Hash Algorithm [22], which has been a standard since 1993, but is included for completeness.

This work was supported in part by NSF under grant CCR-0105677 and by Kodak. A. M. Fiskiran is a Kodak Fellow.

A. Murat Fiskiran and Ruby B. Lee, "Workload Characterization of Elliptic Curve Cryptography and other Network Security Algorithms for Constrained Environments," *Proceedings of the 5th Annual IEEE International Workshop on Workload Characterization (WWC-5), pp. 127-137*, November 2002.

Because elliptic curve cryptography (ECC) algorithms involve mathematical operations that may be new to the architecture community, we describe important ECC characteristics in the following sections.

Table 1 Algorithm set

Class	Algorithm(s)
Public-key	Elliptic-curve Diffie-Hellman key exchange (EC-DHKE) [23], elliptic-curve ElGamal (EC-ElGamal) [15]
Signature	Elliptic-curve Digital Signature Algorithm (EC-DSA) [21]
Symmetric-key	Advanced Encryption Standard (AES) [17]
Hash	Secure Hash Algorithm (SHA) [22]

3. Elliptic curve cryptography (ECC)

3.1. Advantages of ECC

Elliptic curves as mathematical objects have been known and studied since long before digital computers were built, but their application in cryptography has been more recent. In 1985, Victor Miller [16] and Neil Koblitz [10] suggested independently that elliptic curves could be used to perform public-key security functions (e.g. key exchanges, digital signatures). Furthermore, certain unique properties of elliptic curves made them resilient against the types of attacks that were successful against integer-based algorithms. Therefore an elliptic-curve algorithm (e.g. Elliptic-curve Digital Signature Algorithm = EC-DSA) could have the same level of security of a similar integer-based algorithm (e.g. Digital Signature Algorithm = DSA) using much fewer key bits. Table 2 shows the number of key bits necessary to have equivalent levels of security for integer-based algorithms versus elliptic-curve algorithms.

Table 2 Key lengths (in bits) for equivalent security

Integer algorithm (e.g. DSA)	Elliptic-curve algorithm (e.g. EC-DSA)
512	106
768	132
1024	**160**
2048	210
21000	600

For the most commonly used 1024-bit keys for an integer-based algorithm, the elliptic-curve counterpart only requires 160-bit keys for the equivalent security. This is a 7x reduction in the space required to store these keys, or a similar reduction in bandwidth required to transmit these keys over a wireless network. Furthermore, this reduction in the size of data objects allows much faster completion of the algorithms.

Because of these favorable properties, ECC has been incorporated into many security standards. Most importantly, it was added to FIPS 186-2 [21] and IEEE P1363 [9] in 2000.

3.2. Parameters for ECC

Implementation of elliptic curve cryptography involves the selection of a suitable elliptic curve (determined by the coefficients in the elliptic curve equation), the representation of field elements (e.g. a binary field or a prime field), algorithms for field arithmetic and elliptic-curve arithmetic. The standards provide suggestions for the selection of elliptic curves and representation of field elements. FIPS 186-2 [21] recommends a total of ten curves for binary fields: two different curves for each of 163-bit, 233-bit, 283-bit, 409-bit and 571-bit fields. We limit the scope of this study to these ten curves on binary fields, and choose polynomial basis representation for the field elements, which allows faster implementation on programmable processors.

The next section reviews the basic arithmetic operations on polynomials, such as polynomial addition and multiplication, which we will be using frequently in the security algorithms.

4. Polynomial arithmetic

In the polynomial basis representation of a binary field, each field element can be viewed as a polynomial whose coefficients are either 0 or 1. As an example, consider the 163-bit binary field, also denoted GF(2^{163}), recommended in [21]. The following two polynomials $a(x)$ and $b(x)$ are elements of this field:

$$a(x) = x^7 + x^5 + x^3 + x + 1,\ b(x) = x^{162} + x^{100} + x^3$$

In **polynomial addition**, the coefficients of the same powers of x are added component-wise. Since a coefficient can only be 0 or 1, this corresponds to an XOR operation on the coefficients. For example, $a(x)$ is added to $b(x)$ to yield $c(x)$ as:

$$c(x) = a(x) + b(x) = x^{162} + x^{100} + x^7 + x^5 + x + 1$$

(notice that x^3 terms have vanished)

In a software implementation for a 64-bit ISA, each of the polynomials above will fit into three registers, and a single polynomial addition will therefore require three XOR instructions (Algorithm 1).

Similar to addition, **polynomial multiplication** is also component-wise. The key difference is that multiplication may produce a product polynomial that is longer than the multiplicands. Whenever the product is a polynomial of degree greater than 162, it needs to be reduced by the irreducible polynomial $p(x) = x^{163} + x^7 + x^6 + x^3 + 1$ that is specified in [21].

Algorithm 1[1] Polynomial addition

INPUT: Binary polynomials $a(x)$ and $b(x)$ of degree at most 162

OUTPUT: $c(x) = a(x) + b(x)$

1. For i from 0 to 2 do
   1.1. $c[i] = a[i] \oplus b[i]$
2. Return $c(x)$

Algorithm 2[2] Shift-and-add polynomial multiplication

INPUT: Binary polynomials $a(x)$ and $b(x)$ of degree at most 162

OUTPUT: $c(x) = a(x)b(x) \bmod p(x)$

1. If $a_0 = 1$ then $c(x) = b(x)$; else $c(x) = 0$;
2. For i from 1 to 162 do
   2.1. $b(x) \leftarrow b(x)x \bmod p(x)$
   2.2. If $a_i = 1$ then $c(x) = c(x) + b(x)$
3. Return $c(x)$

Algorithm 3 Comb method with windows of width = 4 for polynomial multiplication

INPUT: Binary polynomials $a(x)$ and $b(x)$ of degree at most 162

OUTPUT: $c(x) = a(x)b(x)$

1. Compute $T_u = u(x)b(x)$ for all polynomials $u(x)$ of degree at most 3.
2. $c(x) = 0$.
3. For k from 15 downto 0 do
   3.1. For j from 0 to 3 do
      Let $u = (u_3,u_2,u_1,u_0)$ where u_i is bit $(4k+i)$ of $a[j]$.
      Add T_u to $(c[5],c[4],\ldots,c[j])$.
   3.2. If k != 0 then $c(x) = c(x)x^4$.
4. Return $c(x)$.

Algorithm 4[3] Polynomial reduction (adapted from [8] for 64-bit datapath).

INPUT: Binary polynomial $c(x)$ of degree at most 324.

OUTPUT: $c(x) \bmod p(x)$, where $p(x) = x^{163} + x^7 + x^6 + x^3 + 1$.

1. For i from 5 downto 3 do
   1.1. $t = c[i]$.
   1.2. $c[i\text{-}3] = c[i\text{-}3] \oplus (t << 29) \oplus (t << 32) \oplus (t << 35) \oplus (t << 36)$.
   1.3. $c[i\text{-}2] = c[i\text{-}2] \oplus (t >> 28) \oplus (t >> 29) \oplus (t >> 32) \oplus (t >> 35)$.
2. $t = c[5]$ & 0xFFFFFFF800000000.
3. $c[0] = c[0] \oplus (t >> 28) \oplus (t >> 29) \oplus (t >> 32) \oplus (t >> 35)$.
4. $c[2] = c[2]$ & 0x00000007FFFFFFFF.
5. Return $(c[2],c[1],c[0])$.

Algorithm 5 Table lookup method for polynomial squaring

INPUT: Binary polynomial $a(x)$

OUTPUT: $c(x) = a^2(x)$

1. *Precomputation*: For each byte $v = (v_7,v_6,\ldots,v_1,v_0)$, compute the 16-bit quantity $T(v) = (0, v_7, 0, v_6, \ldots, 0, v_1, 0, v_0)$.
2. For i from 0 to 5 do
   2.1. Let $a[i] = (u_7,u_6,u_5,u_4,u_3,u_2,u_1,u_0)$, where each u_j is a byte.
   2.2. $c[2i] = (T(u_1),T(u_0))$, $c[2i+1] = (T(u_3),T(u_2))$.
3. Return $c(x)$.

Algorithm 6[4] Extended Euclidean Algorithm (EEA) for polynomial inversion

INPUT: Binary polynomial $a(x)$, $a(x) \neq 0$

OUTPUT: $a^{-1}(x) \bmod p(x)$

1. $b(x) = 1$, $c(x) = 0$, $u(x) = a(x)$, $v(x) = p(x)$.
2. While degree$(u(x)) \neq 0$ do
   2.1. j = degree$(u(x))$ – degree$(v(x))$.
   2.2. If $j < 0$ then $u(x) \leftrightarrow v(x)$, $b(x) \leftrightarrow c(x)$, $j = -j$.
   2.3. $u(x) = u(x) + x^j v(x)$, $b(x) = b(x) + x^j c(x)$
3. Return $b(x)$.

[1] **Nomenclature for algorithm descriptions:** Polynomials are represented using lower-case letters: $a(x)$, $b(x)$, $c(x)$ etc. When addressing the individual 64-bit words of a polynomial, square brackets are used: $a[0]$, $b[1]$, $c[2]$ etc. $a[0]$ represents the lowest-order (least-significant) word of $a(x)$. When addressing the individual bits of a polynomial, a subscript is used: a_0, b_{32}, c_{162} etc. The bit a_0 represents the least-significant bit of $a(x)$, and a_{162} represents the most-significant bit. The operator $\oplus$ represents an XOR operation. When used, $p(x)$ denotes the irreducible polynomial generating the field. For GF(2^{163}), $p(x) = x^{163} + x^7 + x^6 + x^3 + 1$.

[2] In Step 2.1, $b(x)x \bmod p(x)$ can be easily computed in two steps as follows. First shift $b(x)$ to the left by one bit (which corresponds to multiplication by x); second, check the 163rd bit of the shifted result, and add $p(x)$ to it if this bit is one (which corresponds to modular reduction).

[3] The operator << denotes logical left shift, >> denotes logical right shift, & denotes bitwise AND.

[4] $a^{-1}(x)$ denotes the multiplicative inverse of $a(x)$, such that $a^{-1}(x)a(x) \bmod p(x) = a(x)a^{-1}(x) \bmod p(x) = 1$. The operator $\leftrightarrow$ denotes a swap of the two values on either side.

Algorithm 7 Modified Almost Inverse Algorithm (MAIA)[5] [8,23] for polynomial inversion

INPUT: Binary polynomial $a(x)$, $a(x) \neq 0$

OUTPUT: $b(x) \in GF(2^k)$ and $t \in [0, 2k-1]$ such that $b(x)a(x) \equiv x^t \bmod p(x)$

1. $b(x) = 1$, $c(x) = 0$, $u(x) = a(x)$, $v(x) = p(x)$, $t = 0$.
2. While x divides $u(x)$ do
   2.1. $u(x) = u(x) / x$, $c(x) = c(x)\, x$, $t = t + 1$.
3. If $u(x) = 1$, return $(b(x), t)$.
4. If degree($u(x)$) < degree($v(x)$) then $u(x) \leftrightarrow v(x)$, $b(x) \leftrightarrow c(x)$.
5. $u(x) = u(x) + x^j v(x)$, $b(x) = b(x) + c(x)$.
6. Go to Step 2.

The simplest algorithm for polynomial multiplication is the shift-and-add algorithm (Algorithm 2). It is presented for illustrative purposes because of its simplicity. The second algorithm, Algorithm 3, is described in [8] and is significantly faster than the shift-and-add algorithm but requires more storage for the table lookups involved.

Algorithm 2 differs from Algorithm 3 in that it does not perform reduction along with multiplication. In Algorithm 2, reduction happens in each iteration in Step 2.1. Therefore, a separate **polynomial reduction** (Algorithm 4) needs to follow Algorithm 3 each time it is used.

Polynomial squaring can be performed using Algorithms 2 or 3 to multiply the input polynomial by itself. However a dedicated algorithm for polynomial squaring gives faster results. Algorithm 5 is a table-lookup based method and it exploits the linearity of the squaring operation in binary fields. If $a(x) = \sum_{i=0}^{162} a_i x^i$, then $a^2(x) = \sum_{i=0}^{162} a_i x^{2i}$. This operation corresponds to inserting zeros between the consecutive bits in the binary representation of $a(x)$. This is facilitated by using a 512-byte table that is precomputed to hold the 16-bit squares of each 8-bit polynomial [8].

For **polynomial inversion**, we present two algorithms: Extended Euclidean Algorithm (EEA) [15] and the Modified Almost Inverse Algorithm (MAIA) [8,23][5]. These are summarized as Algorithm 6 and Algorithm 7 respectively. EEA is a basic but slow algorithm that we provide for illustration, whereas the MAIA (and similar variants of the Almost Inverse Algorithm) is typically used in optimized implementations. While EEA is a direct extension of the basic Euclidean algorithm used for integers, the reader is referred to [8] and [23] for the details of MAIA.

[5] Almost Inverse Algorithm was originally described in [23]. In this study, we use a modified version of that algorithm called Modified Almost Inverse Algorithm, which is described in [8].

5. Diffie-Hellman key exchange

This section focuses on the elliptic-curve Diffie-Hellman key exchange as an example to illustrate the differences of elliptic-curve algorithms from integer algorithms. We first review the integer version.

5.1. Diffie-Hellman key exchange (integer)

Diffie-Hellman key exchange (DHKE) [3] is used to establish a shared key between two parties over a public channel. It is based on the multiplicative group Z_p^* of integers, where p is a prime. Figure 1 shows how a secret key, K, is agreed upon between Alice and Bob by exchanging two quantities a_T and b_T publicly. We assume that the two parties Alice and Bob have agreed on p and α values in advance, where α is a primitive element of the group Z_p^*.

Step	Alice		Bob
1	Choose random a $a \in [2, p-2]$		Choose random b $b \in [2, p-2]$
2	Compute a_T $a_T = \alpha^a \bmod p$		Compute b_T $b_T = \alpha^b \bmod p$
3	Send a_T Receive b_T	$a_T \rightarrow$ $\leftarrow b_T$	Receive a_T Send b_T
4	Compute key K $K = (b_T)^a \bmod p$ $= \alpha^{ab} \bmod p$		Compute key K $K = (a_T)^b \bmod p$ $= \alpha^{ab} \bmod p$

Figure 1 Diffie-Hellman key exchange (integer)

The security of the integer Diffie-Hellman is based on the Discrete Logarithm problem: given p, α and a_T, it is computationally infeasible to compute a (for sufficiently large p). Therefore, even though an eavesdropper may capture the intermediate values a_T and b_T as they are exchanged over the public channel, neither a nor b will be exposed, and therefore the final key K remains known only to Alice and Bob.

5.2. Elliptic-curve Diffie-Hellman key exchange

Elliptic-curve Diffie-Hellman key exchange (EC-DHKE) is similar to the integer version except that it uses the points on an elliptic-curve rather than integers (Figure 2). We assume that Alice and Bob have previously agreed on a binary field GF(2^k), a common elliptic curve E with suitable coefficients[6], and a base point $P=(x,y)$, which lies on E and has order n.

[6] An elliptic curve must have certain properties to be suitable for cryptographic use. A good overview of these is provided in [1].

Step	Alice		Bob
1	Choose random a $a \in [2, n-1]$		Choose random b $b \in [2, n-1]$
2	Compute A_T $A_T = P \times a$		Compute B_T $B_T = P \times b$
3	Send A_T Receive B_T	$A_T \rightarrow$ $\leftarrow B_T$	Receive A_T Send B_T
4	Compute key K $K = B_T \times a$ $= P \times a \times b$		Compute key K $K = A_T \times b$ $= P \times a \times b$

Figure 2[7] Elliptic-curve Diffie-Hellman key exchange

Table 3 Comparison of integer Diffie-Hellman and elliptic-curve Diffie-Hellman key exchanges

	DHKE	EC-DHKE
Group	Integers in Z_p^*	Points on the elliptic curve E
Base object	$\alpha \in Z_p^*$	Elliptic curve point $P \in E$
Primary operation	Exponentiation ($\alpha^a \bmod p$)	Point multiplication ($P \times a$)
Key length[8]	1024 bits	160 bits

The security of the EC-DHKE is based on the elliptic-curve Discrete Logarithm problem: given GF(2^k), E, P, A_T, it is computationally infeasible to compute a (for sufficiently large k and n). Unlike the integer discrete logarithm problem, the elliptic curve discrete logarithm problem has no known sub-exponential time solutions (for a well-chosen set of system parameters[6]). Accordingly, a given level of security can be achieved with a k smaller than the number of bits required to encode the p in the integer Diffie-Hellman key exchange. Table 3 provides a comparison of the basic features of 1024-bit integer Diffie-Hellman and 160-bit elliptic-curve Diffie-Hellman key exchanges.

6. Point multiplication on elliptic curves

Steps 2 and 4 in Figure 2 involve operations on the elliptic curve points. The operator $\times$ denotes the multiplication of an elliptic curve point by a field element. This multiplication is computed by a sequence of doublings and additions of the elliptic curve point (akin to the shift-and-add chains that were used for multiplication of two polynomials). Consider:

$A_T = P \times a$, where $a = (1001) = x^3 + 1$, then

$$A_T = \underbrace{P + P + \ldots + P}_{9 \text{ times}}$$

Or, equivalently

$$A_T = (((P \times 2) \times 2) \times 2) + P$$

The point addition and point doubling operations are described in Algorithms 8 and 9 respectively. Both point addition and point doubling operations reduce to a series of polynomial operations (Table 4), which we have described in Section 4.

Algorithm 8[9] Adding two distinct points on an elliptic curve

INPUT: Elliptic curve points $P = (x_1, y_1)$ and $Q = (x_2, y_2)$, $P \neq Q$

OUTPUT: $R = P + Q = (x_3, y_3)$

1. Compute $\theta = \dfrac{y_2 + y_1}{x_2 + x_1}$.
2. Compute $x_3 = \theta^2 + \theta + x_1 + x_2 + a$.
3. Compute $y_3 = \theta(x_1 + x_3) + x_3 + y_1$.
4. Return (x_3, y_3).

Algorithm 9[9] Doubling a point on an elliptic curve

INPUT: Elliptic curve point $P = (x, y)$

OUTPUT: $R = P + P = (x_3, y_3)$

1. Compute $\theta = x + \dfrac{y}{x}$.
2. Compute $x_3 = \theta^2 + \theta + a$.
3. Compute $y_3 = x^2 + (\theta + 1)x_3$.
4. Return (x_3, y_3).

[7] **Nomenclature for Figure 2:** a and b are scalars. A scalar is an element of the field GF(2^k). As explained in Section 4, addition and multiplication of two scalars is polynomial addition and multiplication respectively. P, A_T and B_T are points on the elliptic curve. Each point on the elliptic curve is determined by its two coordinates. Example: $P = (x,y)$. The coordinates of points are elements of the field GF(2^k). Addition of two points on the elliptic curve is called *point addition* and it is performed with the sequence of formulas described in Section 6. The result of such a point addition is another point on the curve. The operator $\times$ denotes point multiplication. Example: $A_T = P \times a$, whereby the point P on the elliptic curve is added to itself a times. The result A_T is another point on the curve.

[8] Key length in bits for equivalent security. Also see Table 2.

[9] In the equations, a is the coefficient of the x^2 term in the elliptic curve equation. For each binary field, NIST [21] recommends two different elliptic curves: a random curve and a Koblitz curve. Both curves are of the form $y^2 + xy = x^3 + ax^2 + b$, where $a = 1$ and $b \in$ GF(2^k) for random curves, and $a \in \{0,1\}$ and $b = 1$ for Koblitz curves.

Table 4 Number of polynomial operations in point addition and point doubling operations

Polynomial operation	Point doubling	Point addition
Addition	5	9
Multiplication	2	2
Reduction	4	3
Squaring	2	1
Inversion	1	1

Alice		Bob
Choose random a $a \in [2, n-1]$		Choose random $b,\ b \in [2, n-1]$
Compute A_T $A_T = P \times a$		Compute B_T $B_T = P \times b$
Send A_T Receive B_T	$A_T \rightarrow$ $\leftarrow B_T$	Receive A_T Send B_T
Choose random k $k \in [2, n-1]$		
Compute pair (C_1, C_2) $= (P \times k, P_m + k \times B_T)$		
Send (C_1, C_2)	$(C_1, C_2) \rightarrow$	Receive (C_1, C_2)
		Compute $P_m =$ $C_2 - b \times C_1$

Figure 3 Elliptic-curve ElGamal

7. EC-ElGamal, EC-DSA, AES and SHA

Elliptic-curve ElGamal (EC-ElGamal) (Figure 3) is the elliptic-curve analog of the integer ElGamal algorithm described in [4]. It is used to securely transmit the coordinates of the point P_m from Alice to Bob (assume that the original plaintext m is embedded in P_m). We assume that Alice and Bob have previously agreed on a binary field GF(2^k), a common elliptic curve E with suitable coefficients[6], and a base point $P=(x,y)$, which lies on E and has order n.

Elliptic-curve Digital Signature Algorithm (EC-DSA) is based on the ElGamal algorithm and has three different segments: key generation, signature generation and signature verification. These steps are summarized in Figure 4, where Alice signs the message m and Bob verifies the signature.

The Advanced Encryption Standard (AES) became the new U.S. federal standard for block encryption in November 2001 [17]. It can encrypt and decrypt 128-bit, 196-bit and 256-bit blocks. The key length is also variable and can be 128 bits, 196 bits or 256 bits. Compared to the DES/3DES [19] algorithms it is intended to replace, AES is designed to have exceptionally fast software implementations and small code size. In addition, its small memory requirements make it suitable for constrained environments such as smartcards.

Key generation (by Alice)	
1.	Choose random $a \in [2, n-1]$
2.	Compute the intermediate point A_T $A_T = P \times a$ Alice's private key = a Alice's public key = (E, P, A_T)
Signature generation (by Alice)	
1.	Choose random $k \in [2, n-1]$
2.	Compute $P \times k = (x_1, y_1)$ and $r = x_1 \bmod n$ (if $r = 0$, go to Step 1)
3.	Compute $k^{-1} \bmod n$
4.	Compute $s = k^{-1}(\mathrm{SHA}(m) + ar) \bmod n$ (if $s = 0$, go to Step 1) Signature for $m = (r,s)$
5.	Send (r,s)
Signature verification (by Bob)	
	Receive (r,s)
1.	Compute $c = s^{-1} \bmod n$ and SHA(m)
2.	Compute $u_1 = \mathrm{SHA}(m)c \bmod n$ and $u_2 = rc \bmod n$
3.	Compute $P \times u_1 + A_T \times u_2 = (x_o, y_o)$ and $v = x_0 \bmod n$
4.	Accept signature if $v = r$

Figure 4[10] Elliptic-curve Digital Signature Algorithm

Detailed descriptions of the AES algorithm can be found in [2,6] including some optimizations. We report our findings in workload characterization for two AES implementations. The first is the reference implementation, the second is a table-lookup based optimized implementation, both described in [2].

The Secure Hash Algorithm (SHA) is part of the Secure Hash Standard SHS [22]. It is a message-digest function that reads a variable-length input and produces a 160-bit hash of the input. The function that computes the hash is one-way, meaning that it is computationally infeasible to search for an input that evaluates to a given hash value. Because of this one-way property, SHA is used in the Digital Signature Standard [21] to increase efficiency. Instead of signing a long message, only the hash of the message is signed.

[10] SHA(m) represents the hash of the message m computed with the Secure Hash Algorithm described in Section 7 and in [22].

8. Methodology and discussion of results

8.1 Methodology

All the algorithms are coded and optimized in assembly using simple RISC instructions that execute in a single cycle. Assembly coding minimizes the code size, which is important for constrained environments. The RISC ISA has a 64-bit datapath and 32 general integer registers. Of these, R31 is used as stack pointer, R30 as frame pointer. R0 is hardwired to zero; any value written to it is discarded. Only 22 simple instructions are actually used, and these are listed in Table 5. Since these instructions are a subset of the PLX processor, we use the PLX architectural testbed and tools for the simulations and workload characterization [11,12,13,14].

Table 5 Instructions in the basic RISC ISA grouped by instruction classes

Arithmetic	Logical	Shift

`add`
`addi`
`sub`
`subi`
`loadi`
	`and`
	`andi`
	`or`
	`xor`
	`not`
		`sll`
		`slli`
		`srl`
		`srli`
		`sra`
		`srai`

Unconditional branch	**Conditional branch**	**Load/Store**
`call` `ret`	`beqz` `bnez`	`load` `store`

In Table 7, we report instruction frequencies for each of the polynomial operation algorithms described in Section 4. We then select two subsets of these algorithms, to achieve a basic implementation (Setting I) and an optimized implementation (Setting II) of the ECC algorithms (see Table 6). Setting I simulates the simpler algorithms for each polynomial operation. Shift-and-add method (Algorithm 2) is used for multiplication and squaring, and extended Euclidean algorithm (Algorithm 6) is used for inversion. Setting II uses the optimized algorithms. Here, comb method (Algorithm 3) is used for multiplication, Algorithm 4 is used for reduction, table-lookup method (Algorithm 5) is used for squaring, and the modified Almost Inverse Algorithm (Algorithm 7) is used for inversion. In both Settings I and II, Algorithm 1 is used for polynomial addition.

For the elliptic-curve algorithms, we compute workload results for each of the ten curves recommended in [21], although for space reasons, we report results only for key size of 163 bits in Table 7, and for 163 bits and 233 bits in Table 8.

For the cycle counts and speedup calculations, we keep the microarchitecture simple by simulating a single-issue processor. We assume a perfect memory system, where the memory accesses for loads and stores take a single cycle. Instructions are scheduled to eliminate or minimize the pipeline stalls caused by data dependencies.

Table 6 Arithmetic algorithms used in each setting (the numbers refer to the algorithms described in Section 4)

Polynomial operation	I (Basic)	II (Optimized)
Multiplication	2	3
Reduction	N/A	4
Squaring	2	5
Inversion	6	7
Addition	1	1

8.2 Discussion of results

For many of the polynomial operation algorithms in Table 7, the memory instructions (loads and stores) are as frequent as the compute instructions (arithmetic, logical, and shift). Exceptions are the reduction algorithm (Algorithm 4) and the inversion algorithms (Algorithms 6 and 7), which are slightly more compute-intensive. The high percentage of memory instructions (25% to 49%) is due to two factors: (1) the function call overhead, which involves saving and restoring register states at each function call, and (2) the large size of the polynomials, which cannot be kept in the register file during computations and must be written to and read from memory frequently. Arithmetic instructions have over 30% share for Algorithms 1, 2 and 6, while shift instructions are also prominent in Algorithms 3, 4, 5 and 7 with 19% to 28% share. Of the logical instructions, the `xor` is the most frequent one because it performs the basic polynomial addition operation.

Table 8 shows the speedup attainable by using more optimized software algorithms for the different polynomial operations in the three ECC algorithms. This speedup is huge, from 12 to 17 times faster with software optimization. Otherwise, the instruction class distribution is roughly consistent across different field lengths, as well as different algorithms. For all algorithms and field sizes in Setting I, compute instructions comprise 48% to 51% of the instructions, while memory instructions comprise 31% to 38% of the instructions executed. In Setting II, compute instructions comprise a reduced 34% to 40% of the instructions, while memory instructions comprise a larger 45% to 48% of the instructions executed. Looking more closely, arithmetic instructions decrease from over 30% of the instruction mix in Setting I to under 20% in Setting II. The other instruction classes (logical, shift, branches) keep roughly the same percentages, while stores increase slightly, from Setting I to II. The algorithmic optimizations appear to provide speedup mainly by reducing arithmetic instructions.

Table 7 [11] Instruction class frequencies in Algorithms 1-7

Algorithm	1	2	3	4	5	6	7
Elliptic curve from [21]	BR-163	BR-163	BR-163	BR-163	BR-163	BR-163	BR-163
Class (%)							
Arithmetic	35.29	33.51	10.95	9.16	8.58	33.50	11.66
Logical	5.88	7.30	8.19	23.24	18.96	8.98	25.00
Shift	1.47	7.43	25.60	25.35	18.51	7.90	28.39
Unconditional branch	5.88	4.25	5.32	2.82	6.61	6.94	5.37
Conditional branch	5.88	8.19	0.60	0.70	0.70	9.67	4.68
Load	20.59	23.26	25.22	17.61	24.45	19.14	12.45
Store	25.01	16.06	24.12	21.12	22.19	13.87	12.45
TOTAL	100.00	100.00	100.00	100.00	100.00	100.00	100.00
Compute instructions	42.64	48.24	44.74	57.75	46.05	50.38	65.05
Memory instructions	45.60	39.32	49.34	38.73	46.64	33.01	24.90
Pathlength (thousands)	68.0	175.8	73.2	142.0	714.5	721.0	47.1
Cycles (thousands)	68.0	175.8	73.2	142.0	714.5	721.0	47.1
Speedup	1	1	2.40	1	1	1	15.28

Table 8[12] Instruction class frequencies in elliptic-curve Diffie-Hellman key exchange, elliptic-curve El-Gamal, and elliptic-curve Digital Signature Algorithm

Algorithm	Elliptic-curve Diffie-Hellman				Elliptic-curve ElGamal				Elliptic-curve DSA			
Elliptic curve from [21]	BR-163		BR-233		BR-163		BR-233		BR-163		BR-233	
Simulation setting	I	II	I	II	I	II	I	II	I	II	I	II
Class (%)												
Arithmetic	32.68	15.78	33.59	19.41	32.26	17.90	33.78	20.12	31.70	16.09	30.72	17.40
Logical	7.37	9.55	8.43	9.29	7.38	9.27	8.14	9.47	12.33	13.10	11.82	13.58
Shift	8.35	9.30	7.42	9.85	8.42	8.86	7.04	9.62	7.95	10.14	7.29	9.46
Unconditional branch	4.22	8.20	7.88	6.59	7.36	7.92	8.02	7.3	7.64	6.61	6.45	7.29
Conditional branch	8.41	9.95	8.78	7.99	8.88	7.82	8.97	7.37	8.81	8.10	8.11	7.02
Load	23.17	24.44	22.45	25.14	22.63	23.29	23.72	25.84	19.89	25.33	22.54	26.53
Store	15.80	22.78	11.45	21.73	13.07	24.93	10.33	20.28	11.69	20.62	13.07	18.72
TOTAL	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00
Compute instructions	48.40	34.63	49.44	38.55	48.06	36.03	48.96	39.22	51.97	39.33	49.83	40.44
Memory instructions	38.97	47.22	33.90	46.87	35.70	48.22	34.05	46.11	31.57	45.95	35.61	45.25
Pathlength (millions)	74.6	5.2	184.9	14.4	106.0	6.3	270.6	16.6	136.2	8.9	350.0	19.6
Cycles (millions)	74.6	5.2	184.9	14.4	106.0	6.3	270.6	16.6	136.2	8.9	350.0	19.6
Speedup	1	14.48	1	12.88	1	16.88	1	16.31	1	15.26	1	17.86

[11] The elliptic curves are identified by two letters followed by a three-digit number. The first two letters, BR, indicate that the underlying field is a binary field (B), and the elliptic curve is a random curve (R) as described in Footnote 9. The three-digit number indicates the size of the field. Test input for Algorithm 1 is 1,000 randomly initialized polynomial pairs. Test inputs for Algorithms 2 and 3 are 10 randomly initialized polynomial pairs. Test input for Algorithm 4 is 1,000 randomly initialized oversized (up to 325 bits long) polynomials. Test input for Algorithm 5 is 1,000 randomly initialized polynomials. Test inputs for Algorithms 6 and 7 are single randomly initialized polynomials.

[12] The figures for the elliptic-curve algorithms are for the completion of the entire two-party transaction as illustrated in Figures 2, 3 and 4 respectively. For the elliptic-curve Digital Signature Algorithm, the signed message is a randomly initialized 128-bit block.

Table 9[13] Instruction class frequencies in AES and SHA

Algorithm	AES		SHA
Simulation setting	Basic	w/ Table lookups	Basic
Class (%)			
Arithmetic	29.11	20.35	27.77
Logical	17.61	28.32	26.08
Shift	16.30	34.51	17.55
Unconditional branch	7.09	0.00	8.50
Conditional branch	8.72	0.88	8.56
Load	21.17	15.94	5.56
Store	0.00	0.00	5.98
TOTAL	100.00	100.00	100.00
Compute instructions	63.02	83.18	71.40
Memory instructions	21.17	15.94	11.54
Pathlength (millions)	9.5	1.1	1
Cycles (millions)	9.5	1.1	1
Speedup	1	8.6	1

Table 10 Ratio of instructions executed for BR-233 curve to the instructions for BR-163 curve

Algorithm	EC-DHKE		EC-ElGamal		EC-DSA	
Simulation setting	I	II	I	II	I	II
Arithmetic	2.55	3.43	2.67	2.97	2.49	2.37
Logical	2.83	2.71	2.82	2.70	2.47	2.28
Shift	1.20	1.95	1.14	1.87	1.36	1.05
Unconditional branch	4.63	2.24	2.78	2.44	2.17	2.42
Conditional branch	2.59	2.24	2.58	2.49	2.37	1.90
Load	2.40	2.86	2.68	2.93	2.91	2.30
Store	1.80	2.66	2.02	2.15	2.87	1.99
TOTAL	2.48	2.79	2.55	2.64	2.57	2.20

Table 11 Reduction of instruction executed when optimized algorithms are used (Setting II) instead of the basic algorithms (Setting I)

Algorithm	EC-DHKE		EC-ElGamal		EC-DSA	
Elliptic curve from [21]	BR-163	BR-233	BR-163	BR-233	BR-163	BR-233
Arithmetic	29.99	22.30	30.41	27.37	30.06	31.53
Logical	11.17	11.69	13.43	14.01	14.36	15.54
Shift	13.00	9.71	16.06	11.93	11.96	13.76
Unconditional branch	7.45	15.41	15.69	17.91	17.64	15.80
Conditional branch	12.24	14.16	19.16	19.85	16.59	20.64
Load	13.73	11.51	16.40	14.97	11.98	15.17
Store	10.04	6.79	8.85	8.31	8.65	12.47
TOTAL	14.48	12.88	16.88	16.31	15.26	17.86

[13] Test input for AES is a sequence of 1,000 randomly initialized 128-bit blocks. A 128-bit key is used. Test input for SHA is the second test vector given in [22] replicated 1000 times.

Table 12 Ratio of the instruction class frequencies in EC-ElGamal and EC-DSA to EC-DHKE

Algorithm	EC-ElGamal				EC-DSA			
Elliptic curve from [21]	BR-163		BR-233		BR-163		BR-233	
Simulation setting	I	II	I	II	I	II	I	II
Arithmetic	0.99	1.13	1.01	1.04	0.97	1.02	0.91	0.90
Logical	1.00	0.97	0.97	1.02	1.67	1.37	1.40	1.46
Shift	1.01	0.95	0.95	0.98	0.95	1.09	0.98	0.96
Unconditional branch	1.74	0.97	1.02	1.11	1.81	0.81	0.82	1.11
Conditional branch	1.06	0.79	1.02	0.92	1.05	0.81	0.92	0.88
Load	0.98	0.95	1.06	1.03	0.86	1.04	1.00	1.06
Store	0.83	1.09	0.90	0.93	0.74	0.91	1.14	0.86
TOTAL	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table 9 shows that AES and SHA are both compute-intensive. The optimized setting of AES (with table lookups) has 83% compute instructions and 16% memory instructions, while SHA has 71% compute instructions and 11% memory instructions. Of the two simulation settings for AES, arithmetic operations are converted to table lookups in the second setting, which results in a reduced percentage of arithmetic operations, but increased percentage of shift and logical instructions. The shift instructions are used in the table lookups for effective address computations; the logical instructions (primarily `xor`) are used to combine the results of table lookups [2,6].

Table 10, derived from Table 8, shows the increase in the ratio of dynamic instruction counts when the field size is increased from 163 bits to 233 bits. For example, the first value in this table is 2.55, which indicates that the EC-DHKE on the 233-bit field contains 2.55 times as many arithmetic instructions as the EC-DHKE on the 163-bit field, in simulation Setting I. We observe that overall, dynamic instruction counts increase from 2.2 to 2.8 times their previous levels. This shows the expected non-linearity in the complexity of the algorithms. While the field size increases to 1.42 times its previous value (163 bits to 233 bits), the pathlength of the algorithms increases to about 2.5 times its previous value.

Table 11 shows the ratio of the dynamic instruction counts from Setting I to Setting II for each algorithm and field size. This data is useful in understanding where the speedups are achieved. For example, the first value in this table is 29.99, which shows that the EC-DHKE that uses the basic algorithms contains about 30 times as many arithmetic instructions as when it uses the optimized algorithms. Overall, we observe that using the optimized algorithms reduces the dynamic instruction counts by 12 to 17 times. Arithmetic instructions are reduced 22 to 31 times, while other instruction classes are only reduced between 7 to 20 times. This again suggests that the speedups obtained from optimized algorithms come primarily through reductions in the arithmetic instruction counts.

Table 12 compares the relative instruction frequencies of the three elliptic curve algorithms studied. The values in this table are the ratios of the instruction frequencies of EC-ElGamal and EC-DSA to the instruction frequencies of EC-DHKE, which is the base algorithm because of its simplicity. For example, the first value in this table is 0.99, which shows that the frequency of the arithmetic instructions in EC-ElGamal is 0.99 times the frequency of the arithmetic instructions in EC-DHKE under Setting I on a 163-bit field. In general, the ratios across all instruction classes are close to 1 for both EC-ElGamal and EC-DSA. This is because of the similarity of the algorithms; they all are based on elliptic-curve point multiplication. The only significant divergence is in the share of the logical instructions for EC-DSA, which is about 1.5 times as high as in EC-DHKE. This is because the EC-DSA includes SHA computations, which drives up the logical instruction count. As Table 9 indicates, SHA is heavily reliant on logical instructions.

9. Conclusions

The contributions of this paper are: (1) a selection of cryptography algorithms suitable for constrained environments, (2) a description of the operations used by Elliptic Curve Cryptography algorithms, (3) a characterization of the instructions executed by these algorithms, and (4) demonstration that a simple processor is sufficient. We show the operations and instructions needed by elliptic-curve Diffie-Hellman key exchange, elliptic-curve ElGamal, elliptic-curve Digital Signature Algorithm, elliptic-curve arithmetic operations, the Advanced Encryption Standard, and the Secure Hash Algorithm. The importance of these algorithms is verified by the fact that all of them are either standards by themselves, or are part of a larger standard.

For the elliptic curve algorithms, we focused our implementations on binary fields using polynomial basis representation. We compute workload results for the

settings recommended by NIST in [21]. We also described some of the arithmetic algorithms used to implement elliptic-curve operations and presented instruction frequency distributions for each of these algorithms.

We show that these powerful and mathematically complicated algorithms can be implemented very efficiently using a simple RISC-style processor with only 22 single-cycle instructions, implemented by an ALU (Arithmetic Logic Unit) and a shifter. An integer multiplier is not needed.

For future work, we plan to expand our algorithm set to include other elliptic-curve algorithms, different block ciphers, signature algorithms and hash functions. We will also expand our ECC results to include results for prime fields, using bases other than polynomial bases, and different coordinate systems such as projective coordinates.

References

1. Araki, Kiyomichi, Takakazu Satoh, and Shinji Miura, "Overview of Elliptic Curve Cryptography," *Public Key Cryptography, H. Imai and Y. Zheng, eds., pp. 29-48*, Springer-Verlag, 1998.
2. Daemen, J., V. Rijmen, "AES Proposal: Rijndael," *AES submission to NIST*, 1998.
3. Diffie, Whitfield, M. E. Hellman "New Directions in Cryptography," *IEEE Transactions on Information Theory - IT-22, no. 6, pp. 644-654*, November. 1976.
4. ElGamal, T., "A Public-key Cryptosystem and a Signature Scheme based on Discrete Logarithm," *IEEE Transactions on Info Theory, 31:469-472*, 1985.
5. Elsen, I., F. Hartung, U. Horn, M. Kampmann and L. Peters, "Streaming Technology in 3G Mobile Communication Systems," *IEEE Computer, vol. 34, no. 9, pp. 46-52*, September 2001.
6. Fiskiran, A. Murat and Ruby B. Lee, "Performance Impact of Addressing Modes on Encryption Algorithms," *Proceedings of the International Conference on Computer Design (ICCD 2001), pp. 542-545*, September 2001.
7. Hachez, G., F. Koeune, and J. Quisquater, "cAESar results: Implementation of Four AES Finalists on Two Smart Cards", *2nd AES conference*, 1999.
8. Hankerson, D., J. Hernandez, and A. Menezes, "Software Implementation of Elliptic Curve Cryptography Over Binary Fields," *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems (CHES 2000)*, 2000.
9. IEEE P1363 Home Page, Standard Specifications For Public-Key Cryptography, http://grouper.ieee.org/groups/1363/.
10. Koblitz, Neal, "Elliptic Curve Cryptosystems," *Mathematics of Computation, vol. 48, no. 177, pp. 203-209*, 1987.
11. Lee, Ruby B., A. Murat Fiskiran, "PLX: A Fully Subword-Parallel Instruction-Set Architecture for Fast Scalable Multimedia Processing," *Proceedings of the 2002 IEEE International Conference on Multimedia and Expo (ICME 2002), pp. 117-120*, August 2002.
12. Lee, Ruby B., et al., "PLX Project at Princeton University," http://palms.ee.princeton.edu/plx.
13. Lee, Ruby B., et al., Princeton Architecture Laboratory for Multimedia and Security (PALMS), http://palms.ee.princeton.edu.
14. Lee, Ruby B., A. Murat Fiskiran, Zhijie Shi and Xiao Yang, "Refining Instruction Set Architecture for High-Performance Multimedia Processing in Constrained Environments," *Proceedings of the 13th International Conference on Application-specific Systems, Architectures and Processors (ASAP 2002), pp. 253-264*, July 2002.
15. Menezes, Alfred J., Paul C. Van Oorschot and Scott A. Vanstone, "Handbook of Applied Cryptography," CRC Press, ISBN 0-8493-8523-7, October 1996.
16. Miller, Victor S., "Use of Elliptic Curves in Cryptography," *Lecture Notes in Computer Science, no. 218, pp. 417-426*, Springer-Verlag, 1986.
17. NIST (National Institute of Standards and Technology), "Advanced Encryption Standard (AES) - FIPS Pub. 197," November 2001.
18. NIST, Advanced Encryption Standard Development Effort, http://www.nist.gov/aes.
19. NIST, "Data Encryption Standard (DES) - FIPS Pub. 46," January 1977.
20. NIST, 3rd Advanced Encryption Standard (AES) Candidate Conference, April 2000, http://csrc.nist.gov/encryption/aes/round2/conf3/aes3conf.htm.
21. NIST, "Digital Signature Standard (DSS) – FIPS Pub. 186-2," February 2000.
22. NIST, "Secure Hash Standard (SHS) – FIPS Pub. 180-1," 1995.
23. Schroeppel, R., H. Orman, S. O'Malley and O. Spatscheck, "Fast Key Exchange with Elliptic Curve Systems," *Advances in Cryptology - CRYPTO'95, Lecture Notes in Computer Science, pp. 43-56*, Springer-Verlag, 1995.