Memory Performance of Public-Key Cryptography Methods in Mobile Environments

I. Branovic, R. Giorgi, E. Martinelli
University of Siena
Via Roma 56 - Siena, Italy
{branovic,giorgi,martinelli}@dii.unisi.it

Abstract

As an increasing number of Internet hosts are wireless, handheld devices with small memory and strict CPU-latency constraints, the performance of cryptography methods has become critical for high transaction throughput. Elliptic Curve Cryptography (ECC) is emerging as an attractive public-key system for constrained environments, because of the small key sizes and computational efficiency, while preserving the same security level as the standard methods. The memory performance of ECC algorithms was scarcely investigated.

We have developed a set of kernel benchmarks to examine performance of standard and corresponding elliptic curve public-key methods. In this paper, we characterize the operations and their memory impact on performance in Diffie-Hellman key exchange, digital signature algorithm, ElGamal, and RSA public-key cryptosystem, as well as elliptic curve Diffie-Hellman key exchange, elliptic curve digital signature algorithm and elliptic curve El-Gamal algorithm.

We modeled a typical mobile device based on the Intel XScale architecture, which utilizes an ARM processor core and studied the benchmark set on that target. Different possible variations for the memory hierarchy of such basic architecture were considered. We compared our benchmarks with MiBench/Security, another widely accepted benchmark set, in order to provide a reference for our evaluation.

1. Introduction

Cryptography algorithms are split into two categories: private-key (symmetric) and public-key (asymmetric). Internet security protocols (e.g. SSL, IPSec) employ a public-key cryptosystem to exchange private keys and then use faster private-key algorithms to ensure confidentiality of streaming data. In private-key algorithms, communicating parties share a common private key, which is used to transform the original message into a ciphered message. The ciphered message is communicated to another side, and the original message is decrypted by using the same private key. Public-key systems, on the other side, do not require exchange of keys. The public key, known to all, can be used for encrypting messages. However, the resulting ciphertext can only be decrypted using the receiver's private key.

Due to expected advances in cryptanalysis and increases in available computing power, both private and public key sizes must grow over time to offer acceptable security. Table 1 [*NIST00*], [*Blake03*] shows expected key-size growth for various private and public-key cryptosystems: Elliptic Curve Cryptography has attracted attention due to the reduced key size at equivalent levels.

Table 1. Equivalent key size for some cryptosystems.

Public key			private key
ECC key length		RSA key length for approximate equivalent security	length for approximate equivalent security
Prime field	Binary field		
192	163	1024	80
224	233	2048	112
256	283	3072	128
384	408	7680	192
521	571	15360	256

Because of its favourable characteristics, elliptic curve cryptography has been incorporated into two important public-key cryptography standards, FIPS 186-2 [*NIST00*] and IEEE-P1363 [*IEEE1363-00*]. These standards specify how to use elliptic curves over prime fields $GF(p)$ and binary fields $GF(2^m)$; recommended curves have well-studied properties that make them resistant to known attacks. We study elliptic curve analogs of following algorithms for recommended curves in both types of fields:

- Diffie-Hellman key algorithm, used for secure exchange of private keys [*Diffie76*]
- Digital signature algorithm, used for ensuring authenticity of data [*NIST00*]
- El-Gamal algorithm, used for encrypting data [*ElGamal85*].

We compared elliptic curve versions of public-key algorithms with corresponding standard versions. We also included RSA public-key algorithm [*RSA02*], since it is a de-facto standard in this area.

The rest of the paper is organized as follows: in Section 2, we give information necessary for understanding public-key cryptography methods, as well as principles

of ECC. Section 3 gives details on bechmarks used, Section 4 outlines the methodology, while in Section 5 we present a workload characterization of public-key methods with special emphasis on memory perfomance. Section 6 presents related work in this area, and Section 7 concludes.

2. Public-Key Algorithms

In a public-key cryptosystem, the private key is always linked mathematically to the public key. Therefore, it is always possible to attack a public-key system by deriving the private key from the public key. The defense against this is to make the problem of deriving private key as difficult as possible.

2.1. Standard Public-Key Methods

Diffie-Hellman key exchange is used to establish a shared key between two parties over a public channel. Although it is the oldest proposal for eliminating the transfer of secret keys in cryptography, it is still generally considered to be one of the most secure and practical public-key schemes. The security of Diffie-Hellman relays on difficulty of calculating discrete logarithms (given an element α in a finite field F_p and another element y in the same field, find an integer x such that $y = \left|\alpha^x\right|_p$), while it is relatively easy to calculate exponentiation.

Digital signature of a document is a cryptographic means for ensuring the identity of the sender and the authenticity of data. Digital signature of a document is information based on both the document and signer's private key. The National Institute of Standards and Technology (NIST) published the Digital Signature Algorithm (DSA) in the Digital Signature Standard [*NIST00*]. This standard requires use of Secure Hash Algorithm (SHA), specified in the Secure Hash Standard [*NIST95*]. The SHA algorithm takes a long message and produces its 160-bit digest; this method is known as *hashing*. Hash function is hard to invert, which means that given a hash value, it is computationally extremely difficult to find the original message. The message digest is then digitally signed using the private key of the signer; signature can be verified using the sender's private key.

RSA cryptosystem [*RSA02*] is used in the most popular applications, such as SSL, IPSec, e-commerce systems, e-mail systems (PGP, S/MIME); it has practically become the standard for public-key encryption. RSA encryption is based on the fact that the only way of finding the private key is equivalent to factoring an integer, which is computationally impossible if it is long enough. RSA key sizes that today offer acceptable level of security are 1024 bits and longer. Public exponent in common use today is $2^{16} + 1$ (65537), since it improves the efficiency of algorithm.

El-Gamal is a public-key cryptosystem often used as an alternative to RSA. The encryption algorithm is based on dicrete logarithm problem, e.g. finding modular inverses of exponentiations in finite field [*ElGamal85*].

2.2 Elliptic Curve Methods

Unlike standard public-key methods that operate over integer fields, the elliptic curve cryptosystems operate over points on an elliptic curve. The fundamental operation in RSA and Diffie-Hellman is modular integer multiplication. However, the core of elliptic curve arithmetic is an operation called *scalar point multiplication*, which computes $Q = kP$ (point P multiplied by an integer k gives a point Q on the same elliptic curve). The security of ECC lies in the fact that given P and $Q = kP$, it is hard to find k; this problem has similar difficulty as solving discrete logarithm in integer fields, although at the time being this operation seems harder in elliptic curve groups. Consequently, the same level of security is obtained with smaller key sizes compared with standard public-key methods (Table 1). While it is possible to carry out a brute force approach of computing all multiples of P to find Q, by choosing to operate over a large field, for instance binary field $GF(2^{163})$, k is so large that it becomes infeasible to determine k this way. The (large) random integer k is kept as the private key, while the result Q serves as the corresponding public key.

Elliptic curve can be defined over any field, but for cryptographic purposes, we are interested in elliptic curves over finite fields. Finite fields commonly in use in cryptography are prime and binary fields. In binary field, elements can be represented by using polynomial or a normal basis. As it is well-known that polynomial basis yields more efficient software implementations [*Hankerson00*], we used it in developing our benchmarks. Since not every elliptic curve offers strong security properties, standards organizations like NIST have published a set of recommended curves [*NIST00*]. The use of these curves is also recommended for easier interoperability between different implementations of a security protocol. For binary polynomial fields, one random and one Koblitz curve are recommended for key sizes of 163, 233, 283, 409, and 571 bits; for prime fields, for each key size (192, 224, 256, 384, and 521), one curve is recommended.

In the polynomial representation of binary field, each field element can be viewed as polynomial whose coefficients are either 0 or 1. Polynomial addition is defined as simple component-wise XOR of the two polynomials. Polynomial multiplication is also component-wise; the key difference is that multiplication may produce a product polynomial of

degree that is greater or equal to the field size. In such a case, the product needs to be reduced by the irreducible polynomial (usually trinomial or pentanomial), defined in [*NIST00*]. Operations on elliptic curves in binary fields imply using finite field operations. For example, doubling of point in a binary field requires ten finite field operations: two multiplications, one squaring, six additions, and one field inversion [*Menezes01*].

The most time-critical operation in prime field is modular multiplication; among most important improvements is Montgomery algorithm [*Monty85*]. The most time-consuming finite field operation is finding a multiplicative inverse of an element: most often, the extended Euclidean algorithm, or almost inverse algorithm are used [*Fiskiran02*]. Cryptographic algorithms based on discrete logarithm problem can be efficiently implemented using elliptic curves.

3. Benchmark description

We set up a series of kernel benchmarks to cover the cryptographic algorithms mentioned in Section 2. Our benchmarks were written by using *MIRACL* C library procedures for big integer arithmetic [*Miracl02*]. The *MIRACL* library consists of over 100 routines that cover all aspects of multiprecision arithmetic and offers procedures for finite-field elliptic curve operations. All routines have been optimized for speed and efficiency, while at the same time remaining standard, portable C.

Inside the library, a data type called *big* for storing multiprecision integers is defined. It is in the form of a pointer to a fixed length array of digits, with sign and length information encoded in a separate 32-bit integer. Algorithms used to implement arithmetic on the *big* data type are taken from [*Knuth81*]. Montgomery arithmetic [*Monty85*] is used by many of the MIRACL library routines that require extensive modular arithmetic, such as the highly optimized modular exponentiation function *powmod*, and those functions which implement *GF*(*p*) elliptic curve arithmetic.

The benchmarks we setup are listed in Table 4. The use of elliptic curve cryptography also involves choosing a finite field, a specific curve on it, and a base point on the chosen curve. The finite fields and the elliptic curves used in our benchmarks are chosen according to NIST standard [*NIST00*]. Elliptic-curve benchmarks in graphs and tables have the following notation:

<benchmark>.<b | p><three digit number>

The abbreviation *b* denotes an elliptic curve over binary field, while *p* denotes use of a prime field. Three-digit number indicates the size of the field; for example, *b163* denotes the use of $GF(2^{163})$ binary field, while *p384* denotes prime field with binary length of prime *p* equal to 384 (also indicated as ||p||). Therefore, the ec-dh.p192, ec-dh.p224, ec-dh.p256, ec-dh.p384, and ec-dh.p521 represent results of simulating elliptic curve Diffie-Hellman key exchange.

NIST curves over a prime field *GF(p)* are of form:

$$y^2 = x^3 - 3x + b \quad \text{where } b \text{ random}^1$$

while the curves over $GF(2^m)$ are of the form

$$y^2 + xy = x^3 + x^2 + b \quad \text{with } b \text{ random}^2.$$

The elliptic curve methods inside the benchmark use parameter files, for initializing the curve, setting the base point on the curve, and setting the irreducible polynomial for multiplication in binary fields. An example of the structure of prime fields' parameter files *p192.txt, p224.txt, p256.txt, p384.txt, p521.txt* is given in Table 3, while Table 4 gives the typical structure of binary field parameter file. For standard cryptography benchmarks we use the following notation:

<benchmark>.<key_length>

Table 3. Example of the parameter file for a prime field GF(p), ||p||=384.

prime p (dec)	39402006196394479212279040100143613805079739270465446667948293404245721771496870329047266088258938001861606973112319
curve term b (hex)	b3312fa7e23ee7e4988e056be3f82d19181d9c6efe8141120314088f5013875ac656398d8a2ed19d2a85c8edd3ec2aef
base point x coord. (hex)	aa87ca22be8b05378eb1c71ef320ad746e1d3b628ba79b9859f741e082542a385502f25dbf55296c3a545e3872760ab7
base point y coord. (hex)	3617de4a96262c6f5d9e98bf9292dc29f8f41dbd289a147ce9da3113b5f0b8c00a60b1ce1d7e819d7a431d7c90ea0e5f

Table 4. Example of the parameter file for a binary field GF (2^{283}).

degrees of the irreducible polynomial terms	283 12 7 5 0
curve coefficient b (hex)	27b680ac8b8596da5a4af8a19a0303fca97fd7645309fa2a581485af6263e313b79a2f5
elliptic curve base point coordinate x (hex)	5f939258db7dd90e1934f8c70b0dfec2eed25b8557eac9c80e2e198f8cdbecd86b12053
elliptic curve base point coordinate y (hex)	3676854fe24141cb98fe6d4b20d02b4516ff702350eddb0826779c813f0df45be8112f4

The following benchmarks are representative of commonly used public-key methods:

- *dh* (Diffie-Hellman key exchange). It uses a prime suitable for Diffie-Hellman; generates two 160-bit random primes, calculates shared key and writes it into a file.
- *dssign* (digital signing). It reads private key from file, calculates message digest of a file given as argument, and writes signature into a file.
- *dsverify* (digital signature verification). It reads public key and signature from files and verifies it.

[1] Prime field operations are defined as integer addition and multiplication modulo p.

[2] In case of binary field, an irreducible polynomial is used when the degree of multiplication product polynomial is greater than field size.

Table 2. Our public-key benchmark suite.

Benchmark acronym	*Benchmark name*	*Input set description*	*Input set value*
dh.1024	Diffie-Hellman key exchange	key length	1024
ec-dh	Elliptic curve Diffie-Hellman key exchange	elliptic curve parameter file	b163.txt
dssign	Digitally signing a file	file to sign	input_small.asc
dsverify	Verifying a signature of a file	signed file	input_small.dss
ec-sign	Elliptic curve digitally signing a file	elliptic-curve parameter file, file to sign	b163.txt, input_small.asc
ec-verify	Elliptic curve verifying a signature of a file	elliptic-curve parameter file, signed file	b163.txt, input_small.dss
rsae.1024	RSA encryption with exponent 65537	public key length, small text file	1024, test.asc
rsad.1024	RSA decryption with exponent 65537	private key, small encrypted text file	rsa.key, test.enc
elge	ElGamal encryption	public key length, small text file	1024, test.asc
elgd	ElGamal decryption	private key, small encrypted file	elg.key, test.enc
ec-elg	Elliptic curve El-Gamal key generation, encryption and decryption	elliptic-curve parameter file, small text file	p192.txt, test.asc

- *rsae* (RSA encryption with exponent 2^{16}+1). Reads public key from a file and encrypts file passed as argument. Writes ciphered text into a file.
- *rsad* (RSA decryption with exponent 2^{16}+1). It reads private key from file, reads ciphered content from a file passed as argument and writes decrypted content to a file.
- *elge* (ElGamal encryption). It reads public key from a file and encrypts file passed as argument. Writes ciphered text into a file.
- *elgd* (ElGamal decryption). It reads private key from file, reads ciphered content from a file passed as argument and writes decrypted content to a file.
- *ec-dh* (elliptic curve Diffie-Hellman). It reads parameter file for elliptic curve, generates two 16-bit random primes, calculates shared key and writes it into a file.
- *ec-sign* (elliptic curve digital signature). It reads private key and elliptic curve parameters from files, calculates message digest of file given as argument, and writes signature to a file.
- *ec-ver* (elliptic curve digital signature verification). It reads public key, elliptic curve parameters and signature from files and verifies the signature.
- *ec-elg* (elliptic curve El-Gamal). It generates public-private key pair, encrypts the base point on a curve, and then immediately decrypts it.

4. Methodology

The performance evaluation of our public-key cryptography benchmarks is done using the sim-outorder simulators from ARM version of the SimpleScalar toolset [*Ss97*]. The sim-outorder tool performs a detailed timing simulation of the modeled target. Simulation is execution driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction. The ARM target of the SimpleScalar set supports the ARM7 integer instruction set, with the pipeline and memory system models for the Intel StrongARM SA-1110 and XScale SA-2 microprocessors [*Austin02*]. The simulated processor configuration is modeled after Intel ARM Xscale architecture [*Intel03*], with details of configuration given in Table 5. The benchmark code was compiled using arm-linux gcc cross-compiler, available at SimpleScalar Web site [*Ss02*].

We compared the performance of our benchmarks with MiBench/Security benchmarks [*Guthaus01*] (Table 6), available at SimpleScalar Web site [*Ss02*]. MiBench is a set of commercially representative, freely available embedded programs. It offers different categories of real-world embedded applications, among which is the security category. MiBench security category is primarily based on private-key methods; the only public-key application is PGP encryption/decryption, which uses RSA algorithm for signing messages. Due to problems in execution of PGP decode on SimpleScalar-ARM simulator (unimplemented system calls), we simulated only PGP encoding; we expect that PGP decode will have similar performance [*Guthaus01*]. MiBench is used as a reference point including cryptography applications for embedded/mobile environments. To make the comparison as close as possible, we also used MiBench input files (small ASCII text files) as input for our benchmark simulations, where it was applicable. For reasons of space, we reported only the results for the first and the last key size in case of elliptic curve benchmarks (e.g. b163, b571, p192, p521).

When presenting our benchmark statistics (Table 7), number of source lines includes comments. The library files actually used in each benchmark are individuated, and their source lines counted. Static instruction count is obtained by compiling C source and library files with –S option, which produces assembly files, and by counting number of lines in assembly files. Static executable size is the number of bytes occupied on disk, while all dynamic instruction counts, loads and stores are obtained as sim-outorder simulation statistics.

Table 5. Simulated architecture.

Fetch queue (instructions)	4
Branch prediction	8k bimodal, 2k 4-way BTB
Fetch & Decode width	1
Issue width	1 (in order)
ITLB	32-entry, fully associative
DTLB	32-entry, fully associative
Functional units	1 int ALU, 1 int MUL/DIV
Instruction L1 cache	32k-Bytes, 32-way
Data L1 cache	32k-Bytes, 32-way
L1 cache hit latency	1 cycle
L1 cache block size	32 Bytes
L2 cache	none
Mini-data cache	not modelled
Memory latency (cycles)	24 , 96
Memory bus width (bytes)	4

Table 6. MIBENCH/SECURITY benchmarks.

Benchmark acronym	*Benchmark name*	*Input set description*	*Input set value*
bf.enc	Blowfish encrypt	file to encrypt	input_small.asc
bf.dec	Blowfish decrypt	file to decrypt	output_small.enc
rj.enc	Rijndael encrypt	file to encrypt	input_small.asc
rj.dec	Rijndael decrypt	file to decrypt	output_small.enc
sha	SHA	file to hash	input_small.asc
pgp.enc	PGP encode	small text file	test.asc
pgp.dec	PGP decode	small text file	test.enc

5. Workload Characterization

In this Section, we characterize select benchmarks with particular emphasis on the memory behavior.

In Table 7, static and dynamic figures for standard cryptography algorithms (`dh`, `dssign`, `dsverify`, `rsa`, `elg`) and their ECC equivalent (`ec-dh`, `ec-sign`, `ec-verify`, `ec-elg`) are reported. We considered operations involving same level of security by choosing an appropriate key length, as discussed in Introduction (see Table 1): 1024 bits for standard public-key cryptography, 192 bits for prime field based ECC, and 163 bits for binary field based ECC.

In our characterization, we also included the MiBench/Security suite, in order to provide a direct comparison with a widely known benchmark suite in our experimental setup. In Table 8, we report the same figures for MiBench/Security.

Using ECC does not involve a higher number of dinamically executed instructions (fourth column, Table 7), compared with the same figure for standard cryptography. For `elg` (ElGamal), we should consider the sum of `elge` and `elgd` compared with `ec-elg`. RSA case is discussed below. The number of memory operations is percentually higher for binary-field ECC algorithms than in standard cryptography (Figure 3). In particular, *the number of load operations is always percentually higher. This may mean that a particular care has to be taken for the memory subsystem, when considering the implementation of binary-field ECC algorithms*. This is particularly true for mobile systems such as PDA or wireless phones, where memory could be not very fast and caches have a small size due to power constraints.

In case of prime-field ECC, the percentage of memory operations (and loads) is more similar to standard cryptography methods (Figure 3).

For all ECC methods, the number of load operations is more similar to complex private-key encryption schemes such as `rj` (Rijndael, also adopted as AES, Advanced Encryption Standard).

The number of memory references is higher in standard cryptography than in binary-field ECC (last two columns, Table 7), but a further analysis is needed to see if they really contribute to the total execution time. In fact, in Figure 4 – on the left – where we selected 24 cycles for memory latency and 1Kb for Level-1 Instruction Cache and 1Kb for Level-1 Data Cache, it appears that binary-field ECC algorithms take a longer time to execute than their corresponding standard version. In that graph, we also show the contribution due to memory stall (upper portion of bars). In case of prime-field, again the figures are mostly similar to standard cryptography case: anyway, the total execution time increases much over the standard cryptography case when longer keys are used.

This means that, *even if ECC use a lower number of memory operations, the working set is larger or the locality of instruction and data accesses is somewhat worse than in standard cryptography*.

Both latter problems can be overcome through the use of larger caches. Therefore, we considered a more detailed analysis of the caches. As our goal is to analyze this situation in the case of mobile systems, we setup typical configurations of Xscale processor, with only Level-1 Instruction+Data split caches and no Level-2 cache. In Figure 5, we report the misses per 1000 instructions for all the considered algorithms and for cache sizes from 256-bytes through 32K-bytes. The Misses-Per-Instruction (MPI) metric is useful as it provides a figure that is directly proportional to the CPI (Cycles Per Instruction) contribution due to memory stall [*Kessler91*]. The cache size range is appropriate for our case as the working set size is rather small (as typical in embedded systems applications [*Guthaus01*]). For a 32K-bytes cache size the MPI approaches zero.

To analyze further the reasons of the higher stall time of Figure 4, we also report in Figure 6 a detail of the Data and Instruction MPI in the case of 1-Kbyte caches. This confirms again that the situations with a higher total execution time in Figure 4 are closely related to the higher MPI either for instruction or data caches.

In the case of RSA algorithm, MPI is lower than ECC encryption/decryption schemes for Instruction cache but higher for Data cache. This indicates that RSA uses

code that is more optimized and makes more extensive use of larger or more complex data structures.

From Figures 4 and 5, we can conclude that: I) even if the total number of instructions and memory references is lower for ECC algorithms compared with standard methods, both *Instruction and Data locality matters to ECC performance and appropriate caches should be adopted in order to keep the total execution time at acceptable levels*; for example, in the case of `ec-dh` algorithm with 163 bit key-length on the binary field more than 75% of the execution time is due to memory stall. To reduce the execution time, we should have at least 16K-bytes of instruction cache and 2K-bytes of data cache available for this applications. II) If the constraints of our system design require a slower (lower-power) main memory, the stall time due to memory access could be even higher (Figure 4, right portion, where main memory latency is 96 cycles).

Another interesting comparison is between ECC methods working on 163 bits and those working on 571 bits (for binary field; in the case of prime field this numbers are respectively 192 and 521, see Table 1). As we can see (Figures 4, 5, 6), *the importance of memory stall and thus the importance of appropriate caches is more relevant in the case of binary field rather than in the case of prime field*.

For MiBench/Security, we observe that private-key algorithms (`bf` and rj) have a much higher number of data misses (Figure 6, 1Kbyte cache size), while the public-key based `pgp.enc` nicely compares with same values as the standard cryptography benchmarks that we selected. For Instruction misses, the private-key algorithms compare more directly with ECC algorithms rather than with standard public-key methods.

Table 7. Public-key benchmark statistics (cache size 32 kB, memory latency 24).

Benchmark name	Source lines (application +app_library)	Instruction count		Static executable size (bytes)	Dynamic executable size (bytes)		Loads dynamic	Stores dynamic
		static (application+app_library)	dynamic		text	data		
dh.1024	47/8365	147/20226	165,901,894	1,040,077	126,976	131,072	54,590,385	22,853,261
ec-dh.b163	90/10821	332/30884	71,136,495	1,099,417	176,128	167,936	26,788,056	12,093,928
ec-dh.p192	86/9356	289/21217	78,703,796	1,059,432	147,456	163,840	28,603,873	11,253,834
dssign,1024	203/8522	653/20773	95,408,044	1,196,890	167,936	159,744	36,231,691	16,861,891
dsverify.1024	123/8522	419/20773	101,343,319	1,195,925	155,648	155,648	38,170,154	17,675,779
ec-sign.b163	133/10978	468/31431	80,758,444	1,100,817	176,128	163,840	32,241,048	15,417,012
ec-sign.p192	144/9513	418/24701	84,280,446	1,100,689	155,648	172,032	33,277,790	15,433,828
ec-verify.b163	143/10978	524/31431	92,130,376	1,253,511	200,704	176,128	36,574,617	17,283,606
ec-verify.p192	153/9513	472/25369	92,351,655	1,253,303	192,512	188,416	36,120,344	16,563,182
rsae.1024	73/8365	273/20226	6,445,027	1,045,101	135,168	147,456	2,013,733	848,857
rsad.1024	113/8479	448/20942	55,064,462	1,046,308	159,744	151,552	18,525,278	7,624,237
elge.1024	95/8365	366/20226	58,287,669	1,045,748	143,360	151,552	19,157,821	7,996,009
elgd.1024	88/8365	327/20226	384,535,577	1,043,366	135,168	143,360	127,736,147	53,274,033
ec-elg.b163	96/10821	367/30884	220,968,247	1,099,563	155,648	184,320	84,206,061	28,097,746
ec-elg.p192	94/9356	316/21217	129,476,585	1,059,594	143,360	176,128	38,263,497	17,293,027

Table 8. MIBENCH/SECURITY: Benchmark statistics (cache size 32 kB, memory latency 24).

Benchmark name	Source lines	Instruction count		Static executable size (bytes)	Dynamic executable size (bytes)		Loads dynamic	Stores dynamic
		static	dynamic		text	data		
bf	2,302	7,749	52,412,141	968,691	56,244	102,400	19,971,657	17,391,240
rj	1,773	12,134	30,737,771	998,449	81,920	126,976	13,441,929	4,179,454
sha	269	793	13,540,983	955,216	69.632	106,496	2,281,656	1,236,606
pgp	34,858	73,244	39,106,462	1,451,988	217,088	450,560	8,609,025	4,690,129

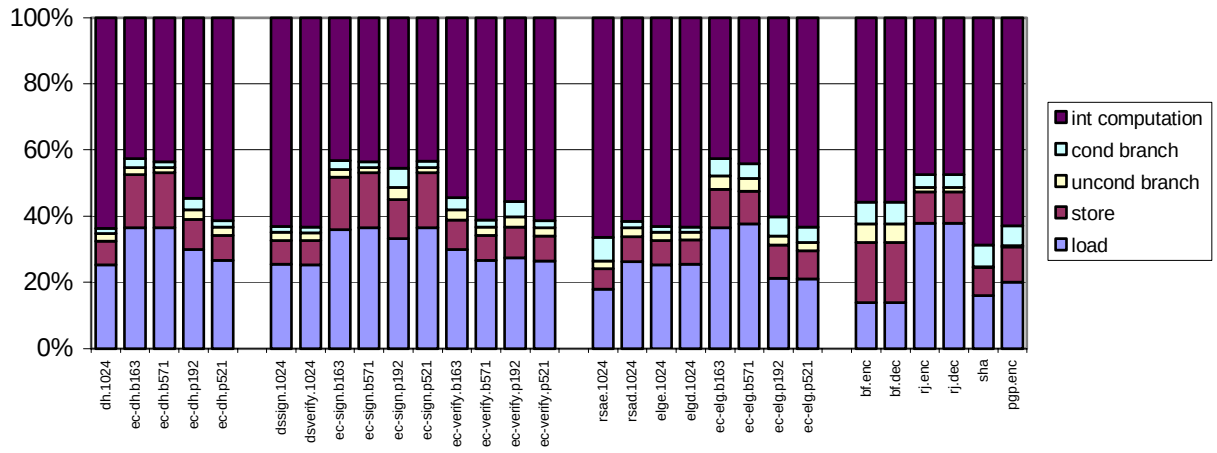

Figure 3. Dynamic instruction class profile for our benchmarks and MIBENCH/SECURITY.

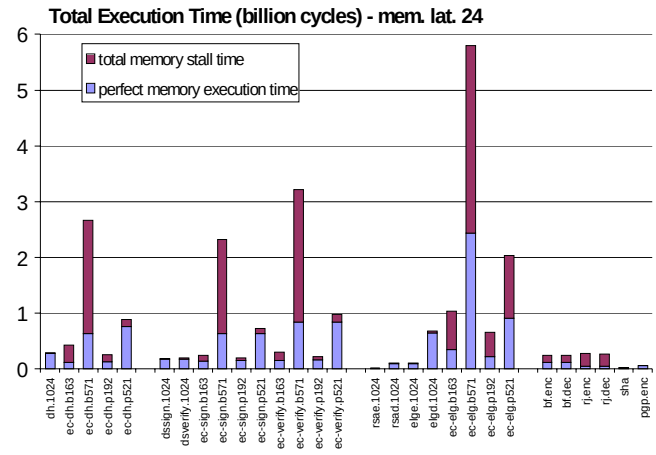

Figure 4. Variation of the total execution time with memory latency (1K-byte Data + 1K-byte Instruction cache).

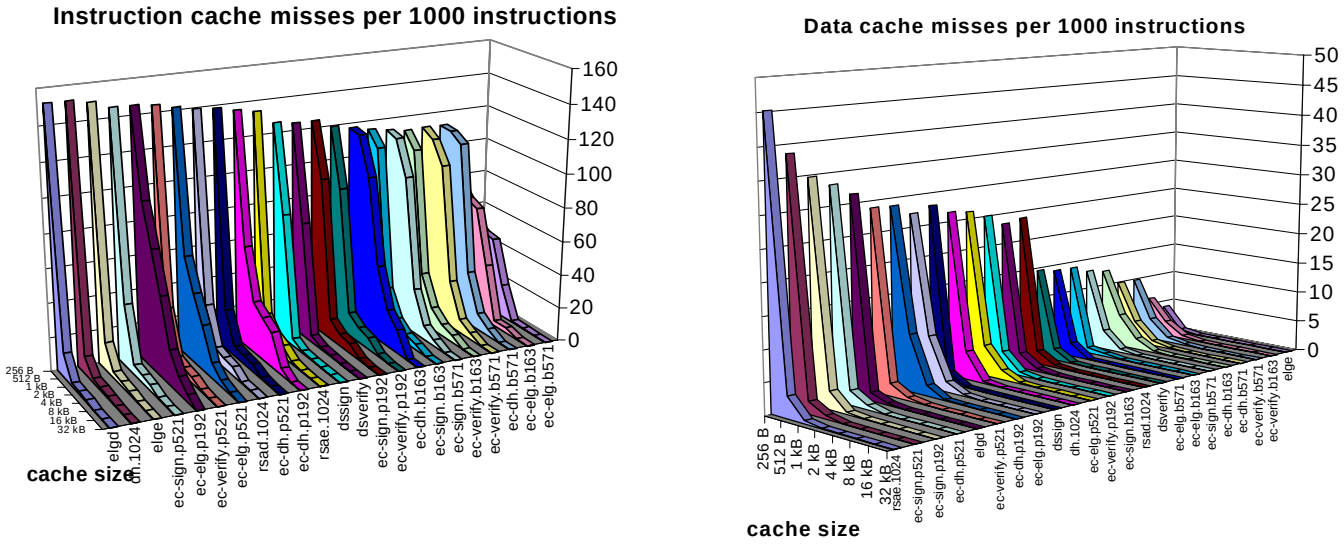

Figure 5. Instruction and data cache misses per 1000 instructions.

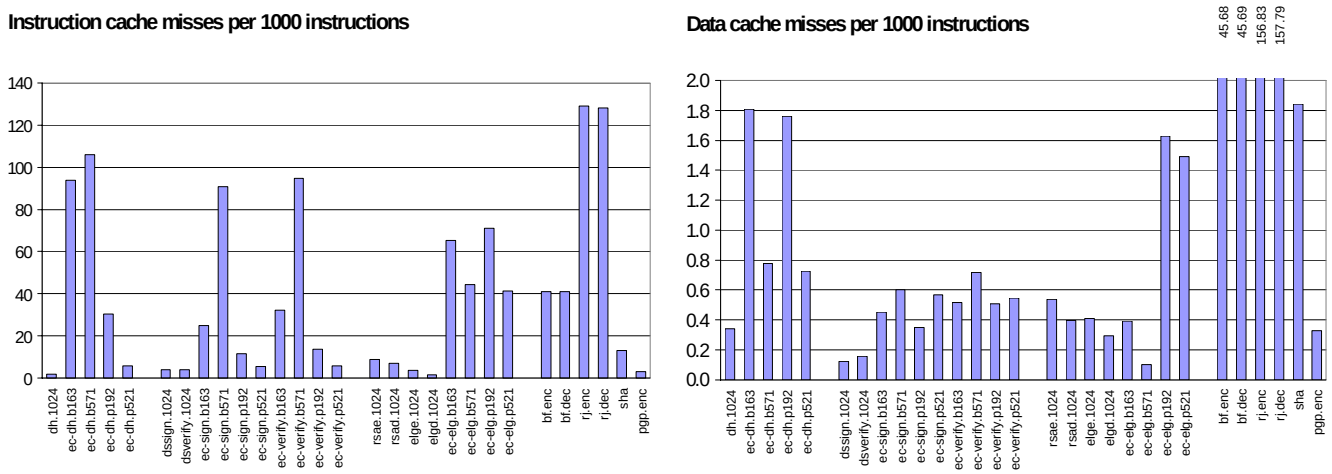

Figure 6. Instruction and data cache misses per 1000 instructions (L1 cache 1+1 KB) and comparison with MiBench.

6. Related Work

There is an intensive research ongoing in improving the efficiency of elliptic curve operations, as well as their performance analysis. A workload characterization of some public-key and private-key algorithms, including their elliptic-curve equivalents for binary polynomial fields is found in [*Fiskiran02*]. They characterize operations in Diffie-Hellman, digital signature, and El Gamal elliptic curve methods, and demonstrate that all these algorithms can be implemented efficiently with a very simple processor. [*Hankerson00*] presents an extensive and careful study of the software implementation of NIST-recommended elliptic curves over binary fields. In [*Gupta02*], the authors give the first estimate of performance improvements that can be expected by adding ECC support in SSL protocol.

In [*Guthaus01*], a set of freely available to researchers, commercially representative benchmarks for embedded systems, called MiBench, is compared with SPEC2000 benchmarks, which characterizes a workload for general-purpose computers. The common characteristics of security applications are low cache miss rate, more than 50% integer ALU operations, and low level of parallelism. In [*Milenkovic03*], MiBench suite and SimpleScalar simulator for ARM target are used for a performance evaluation of typical cache design issues for embedded systems.

7. Conclusions

The main contributions of our paper are: i) setup of kernel benchmark set for studying elliptic curve and standard public-key methods and ii) studying the impact of memory hierarchy in mobile systems.

We found that using ECC does not involve a higher number of dinamically executed instructions. Even if ECC uses a lower number of memory operations, the working set is larger or the locality of instruction and data accesses is worse than in standard cryptography.

Instruction and Data locality matters to ECC performance and appropriate caches should be adopted in order to keep total execution time at acceptable levels. The importance of memory stall and thus the importance of appropriate caches is more relevant in the case of binary field than in the case of prime field.

Acknowledgments

This work is supported by Italian Ministry of Education, University and Research, under subcontracting of project FIRB "Reconfigurable platforms for wideband wireless communications", protocol RBNE018RFY. We are particularly grateful to Todd Austin for his help on the initial setup of our experiments.

References

[*Austin02*] T. Austin, E. Larson, D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modelling", *IEEE Computer* , Volume 35, Issue: 2, Feb. 2002, pp. 59–67.

[*Blake03*] V. Gupta, S. Blake-Wilson, B. Moeller, C. Hawk, "ECC Cipher Suites for TLS", Internet draft, June 2003, http://www.ietf.org/internet-drafts/draft-ietf-tls-ecc-03.txt.

[*Diffie76*] W. Diffie, M. E. Hellman, "New Directions in Cryptography", *IEEE Trans. on Information Theory*, Vol. IT-22, Nov. 1976, pp. 644-654.

[*ElGamal85*] T. ElGamal, "A public key cryptosystem and a signature scheme based on discrete logarithms", *IEEE Trans. on Information Theory*, Vol. 31, 1985, pp. 469-472.

[*Fiskiran02*] A. M. Fiskiran, R. B. Lee, "Workload Characterization of Elliptic Curve Cryptography and other Network Security Algorithms for Constrained Environments", *Proc. of 5th IEEE Workshop on Workload Characterization (WWC-5)*, Nov. 2002, pp. 127-137.

[*Gupta02*] V. Gupta, S. Gupta, S. C. Chang, "Performance Analysis of Elliptic Curve Cryptography for SSL", *WiSe'02*, Atlanta, USA, 2002.

[*Guthaus01*] M. Guthaus, J. Ringerberg, T. Austin, T. Mudge, R. Brown, "MiBench: A free, commercially representative embedded benchmark suite", *Proc. of 4th Workshop on Workload Characterization*, Dec. 2001.

[*Hankerson00*] D. Hankerson, J. Lopez, A. Menezes, "Software Implementation of Elliptic Curve Cryptography over Binary Fields", *Proc. of CHES 2000 Conference*, Springer-Verlag, 2000, pp. 1-24.

[*IEEE1363-00*] IEEE Standard Specifications for Public-Key Cryptography, 1363-2000, IEEE Computer Society, Jan. 2000, http://grouper.ieee.org/groups/1363/

[*Intel03*] Intel Corporation, "The Intel Xscale Microarchitecture Technical Summary", ftp://download. intel.com/design/intelxscale/XscaleDatasheet4.pdf

[*Kessler91*] R. E. Kessler, "Analysis of Multi-Megabyte Secondary CPU Cache Memories", Ph.D. Thesis, Univ. of Wisconsin, Computer Sciences, Tech. Report 31032, 1991.

[*Knuth81*] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Addison-Wesley, 1981.

[*Menezes01*] A. Menezes, *Elliptic Curve Public Key Crypto-systems*, Kluwer Academic Publishers, Boston, USA, 2001.

[*Milenkovic03*] A. Milenkovic, M. Milenkovic, N. Barnes, "A Performance Evaluation of Memory Hierarchy in Embedded Systems", *Proc. of 35th SSST*, Mar. 2003.

[*Miracl02*] Miracl big integer library Web site, http://indigo.ie/~mscott/

[*Monty85*] P. Montgomery, "Modular Multiplication Without Trial Division," *Mathematics of Computation*, 44, Apr. 1985, pp. 519-521.

[*NIST95*] NIST, Secure Hash Standard, FIPS pub180-1, 1995.

[*NIST00*] Digital Signature Standard, National Institute of Standards and Technology, FIPS pub186-2, Jan. 2000.

[*RSA02*] RSA Laboratories' FAQs about Today's Cryptography, http://www.rsasecurity.com/rsalabs/faq

[*Ss97*] D. C. Burger, T. M. Austin, "The SimpleScalar Tool Set, Version 2.0", Tech. Report CS-TR-97-1342, University of Wisconsin-Madison, June 1997.

[*Ss02*] SimpleScalar LLC, http://www.simplescalar.com