A Hardware Architecture for Elliptic Curve Cryptography and Lossless Data Compression

Miguel Morales-Sandoval and Claudia Feregrino-Uribe
National Institute for Astrophysics, Optics and Electronics
Computer Science Department
Luis Enrique Erro No. 1, Sta. Ma. Tonantzintla, Pue, 72840 Puebla, México
{mmorales, cferegrino}@inaoep.mx

Abstract

We present a hardware architecture that combines Elliptic Curve Cryptography (ECC) and lossless data compression in a single chip. Input data is compressed using a dictionary-based lossless data compressor before encryption, then; two elliptic curve cryptographic algorithms can be applied to the compressed data: ECIES for encryption or ECDSA for digital signature. Applying data compression presents three advantages: first, the improvement in the cryptographic module throughput by reducing the amount of data to be encrypted; second, the higher utilization of the available bandwidth if encrypted data is transmitted across a public network and third, the increment of the difficulty to recover the original information. The architecture was described in VHDL and synthesized for a Xilinx FPGA device. The results achieved show that it is possible to combine these two algorithms in a single chip while gathering the advantages of compression and cryptography. This work is novel in the sense that no such algorithm combination has been reported neither a hardware implementation of elliptic curve cryptographic schemes.

1. Introduction

Data compression and cryptography play an important role when transmitting data across a public computer network. While compression reduces the amount of data to be transferred or stored, cryptography ensures that data are transmitted with reliability and integrity. In theory, compression and cryptography are opposite: while cryptography converts some legible data into some totally illegible data, compression searches for redundancy or patterns in data to be eliminated in order to get a reduction of data.

Using a data compression algorithm together with an encryption algorithm, in the correct order, makes sense for three reasons:

- Compressing data before encryption reduces the redundancies that can be exploited by cryptanalysts to recover the original data.
- Compressing data speeds-up the encryption process.
- If encrypted data are transmitted in a computer network, the bandwidth is better utilized.

Data must be compressed before encryption. If it were the opposite case, the result of the cryptographic operation would be illegible data and no patterns or redundancy would be present, leading to very poor or no compression at all.

The approach of combining compression with cryptography has been adopted in some software applications like HiFn [7], PKWare [15], PGP [19] and CyberFUSION [16]. Also, some network security protocols like SLL, SSH and IPSec compress data before encryption as part of the transferring process. PGP uses symmetrical ciphers, CAST-128, IDEA and 3DES for encryption, and RSA for public key cryptography. Messages signed or encrypted are compressed using the ZIP algorithm. The popular PKWare's software, PKZip, encrypts messages for storage or transfer using symmetrical encryption, 256-bit key AES, or asymmetrical encryption, RSA. CyberFUSION, similar to a FTP application, encrypts data using either the DES or 3DES algorithm. Compression is performed using the RLE (Run Length Encoding) algorithm.

HiFn proposed a processor to perform both compression and encryption. Cryptographic symmetrical algorithms supported by this processor are AES and 3DES, and SHA-1 and MD5 for authentication [19]; compression is performed by the LZS (Lempel-Ziv-Stac) [5] algorithm. CISCO offers some hardware and software modules to encrypt and compress incorporated into routers in order to improve the performance of data transmission. Data can be compressed by the LZS algorithm or by the IPPCP compression protocol; the compressed data are encrypted by the AES algorithm with 128, 192 or 256-bit key.

0-7695-2283-1/05 $20.00 © 2005 IEEE

Compression and cryptographic algorithms are expensive in terms of time when they are implemented in general purpose processors (like the ones used in personal computers). When implementing compression algorithms, the search for redundancy implies many complex operations that can not be implemented efficiently with the available instruction set of a general purpose processor. And when cryptographic algorithms are implemented, it is necessary to perform a high amount of mathematical operations between large numbers in a finite field. Again, general purpose processors do not have instructions to support these operations, leading to inefficient implementations. For these reasons, a hardware solution is well suited to implement both kinds of algorithms, especially for real time data processing.

In this paper, we implement public key cryptography instead of symmetrical encryption. Traditionally, public key cryptography has been used only to generate a shared secret value, which is used for bulk encryption. We now consider how public key cryptography performs to encrypt data. Furthermore, we compress data before encryption operations in order to improve the performance of the cipher module. Compression is performed by a dictionary-based lossless data compressor, a variant of the LZ77 algorithm [22], the LZSS [20]. Compressed data are encrypted using Elliptic Curve Cryptography (ECC) [10], implemented schemes are the Elliptic Curve Integrated Encryption Scheme (ECIES) [17] for bulk encryption and the Elliptic Curve Digital Signature Algorithm (ECDSA) [1] for digital signature.

To our knowledge, there is no hardware implementation where lossless data compression and elliptic curve cryptography have been considered jointly, neither a hardware implementation of the ECC schemes.

The remainder of this paper is organized as follows: Section 2 describes the cryptographic schemes implemented in this work, section 3 presents the system architecture and explains how data flow occurs; Section 4 presents the synthesis results and timing for scalar multiplication and data compression. Finally, section 5 concludes this work and presents future directions.

2. Data compression and ECC

ECC is a relatively novel approach for public key cryptography. It uses shorter length keys than other public-key cryptosystems offering the same security level. For example, using a 163-bit offers the same security level that RSA with a 1024-bit key. This implies less space for key storage and faster arithmetic operations. Furthermore, it has been shown in the literature [10] that ECC's security is higher than that provided by RSA, which is the most widely used public key cryptosystem. The LZ77 algorithm is the first proposal for text compression where prior knowledge or statistical characteristics of the symbols are not required. This fact leads to faster compression because a second pass over the data is not required as it occurs in some statistical methods. A second advantage is that the decompression process is easier and faster than the compression one. These two reasons made LZ77 attractive for us to implement it and study it as a competitive lossless data compressor to be used previous the elliptic curve cryptosystem.

An elliptic curve cryptosystem consists of a *7-tuple* $T = (q, a, b, G, n, h)$ where q is the finite field where the elliptic curve is defined, a and b are elements in the finite field that define the elliptic curve equation, G is a point of the elliptic curve and has the property of generating all other points defined by the same elliptic curve, n is the order of the point G and h is the divider of the number of elements of the elliptic curve by n [2].

In this work, we select the two-characteristic finite field F_{2^m}, according to literature, this field leads to more efficient hardware implementations than other finite fields [2]. For F_{2^m} , an elliptic curve is defined as a set of points satisfying equation 1.

$$y^2 + xy = x^3 + ax^2 + b \tag{1}$$

The points of the elliptic curve form a group respect to the *sum* operation. On this group, the discrete logarithm problem is defined as follows: given two points on the curve, say P and Q, find the scalar k such that $kP = Q$. As this problem is considered extremely difficult for special elliptic curves, ECC bases its security on this problem. On the contrary, given the scalar k and a point on the curve P, the operation kP is relatively easy to compute. This operation is called scalar multiplication and it is a critical operation in the cryptographic schemes based on elliptic curves, two of them are implemented in this work. For these schemes, assume that the *7-tuple* T is shared by entities **A** and **B**, d_A and d_B are private keys of entities **A** and **B** respectively and Q_A and Q_B are the public keys of **A** and **B** respectively.

Entity **A** performs the following steps for encrypting a message m_1 for **B**,

1. Select a random number $k \in [1, n-1]$
2. Compute $(x, y) = kQ_B$ and $R = kG$
3. Use a *Key Derivation Function (KDF)* to derive a $(S + M)$-bit key, k_{KDF}, from x
4. Take the S left most bits of k_{KDF} as the S-bit key k_S and encrypt the message. $C = E(m_1, k_S)$
5. Take the M right most bits of k_{KDF} as the M-bit key k_M and compute the m's MAC value. $V = MAC(m_1, k_M)$
6. Send (R, C, V) to **B**

To recover the original message, **B** perform the following steps:

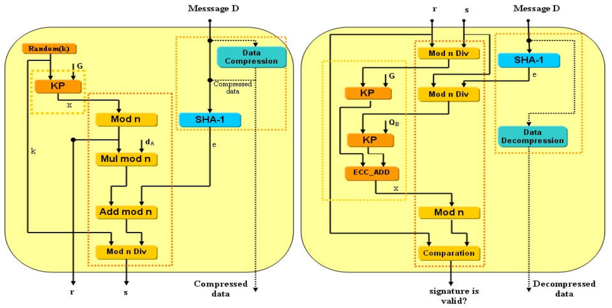

Figure 1. ECDSA

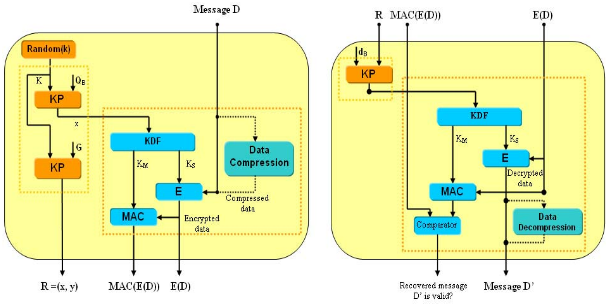

Figure 2. ECIES

1. If R is not a valid elliptic curve point, fail and return.
2. Compute $(x', y') = d_B R$
3. Use a *Key Derivation Function (KDF)* to derive a $(S+M)$-bit key, k_{KDF}, from x'
4. Take the S left most bits of k_{KDF} as the S-bit key k_S and decrypt the message C. $m_1 = E(C, k_s)$.
5. Take the M right most bits of k_{KDF} as the M-bit key k_M and compute the C's MAC value. $V = MAC(C, k_M)$
6. Accept message m_1 as valid if and only if $V = V_1$

The ECDSA works as follows: To sign the message m_1, entity **A** performs the following steps:

1. Select a random number k from $[1, n-1]$
2. Compute $R = kG = (x, y)$ and $r = x \bmod n$. If $r = 0$ go to step 1.
3. Compute $s = k^{-1}(H(m_1) + d_A r) \bmod n$, H is the hash value of the message.
4. The digital signature on message m_1 is the pair (r, s)

Entity **B** can verify the digital signature (r, s) on m_1 performing the following steps:

1. Verify r and s are integers in $[1, n-1]$, if not, the digital signature is wrong. Finish and reject the message.
2. Compute $w = s^{-1} \bmod n$ and $H(m_1)$, H is the hash value of the message.
3. Compute $u_1 = H(m_1)w \bmod n$ and $u_2 = rw \bmod n$
4. Compute $R' = u_1 G + u_2 Q_A = (x', y')$
5. Compute $v' = x' \bmod n$, accept the digital signature if and only if $v' = r$

Block diagrams of the ECDSA and ECIES schemes, showing where data compression occurs, are depicted in figures 1 and 2. In both schemes, support for elliptic curve operations is required. In ECDSA, it is necessary to perform mathematical operations on large integers. In ECIES, the **KDF** module derives a key as a bit string of arbitrary length l by executing the SHA-1 algorithm $l/160$ times. **KDF** is specified in standard ANSI X9.63. The MAC algorithm considered in this work is **HMAC** using a 160-bit k_M key. **HMAC** is specified in ANSI X9.71. One of the symmetrical encryption methods recommended by SEC-1 for ECIES symmetrical encryption is the XORing encryption. This kind of encryption consists in a XOR operation between the key k_S and data. So, the **KDF** module must generate a 160-bit k_M key and a k_S key of the same length that the message to be encrypted/decrypted. Theoretically, we need to know the length of data *a priori* in order to know how many SHA-1 iterations will be executed.

HMAC y KDF, are based on the SHA-1 algorithm [13], which is used in the ECDSA scheme too as the hash function. The SHA-1 algorithm assumes all data is available in order to compute the hash value. In KDF, SHA-1 computes a hash value on fixed size data, but in HMAC and ECDSA, the size is determined by the input message. For a signature generation operation, data are compressed before computing the hash value. For an encryption operation, data are compressed before they are encrypted by the E module. In a signature verification or decryption operation, data are assumed to arrive in a compressed form, so, incoming data are not compressed but decompressed after the cryptographic operations.

In order to outperform a sequential implementation data are processed in each module as they are being processed in previous modules, as a *pipeline* approach. For example, in an encryption operation, data are authenticated as they are been encrypted, that in the same way, data are being encrypted as they are been compressed. Because of arithmetic operations do not depend on partial results in data processing blocks, these can be supported by independent arithmetic units, one for elliptic curve arithmetic and other for modular integer arithmetic. In both schemes, arithmetic operations and data processing can be performed in parallel, as shown in figures 1 and 2.

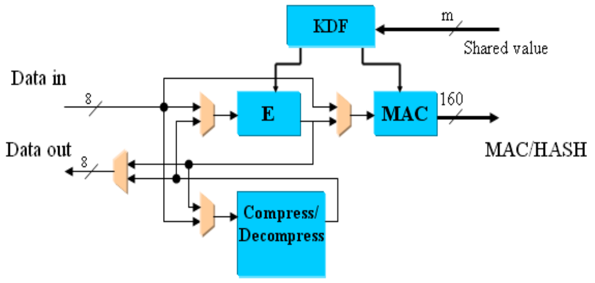

Figure 3. Data processing

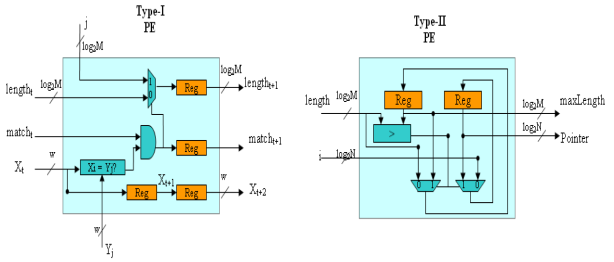

Figure 4. Processing elements

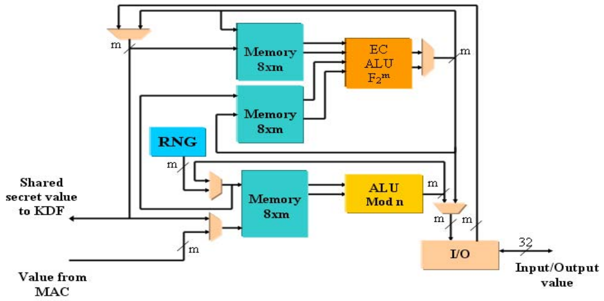

Figure 5. Arithmetic units diagram

3. Architecture of the system

Figure 3 shows how the main modules for data processing in ECIES and ECDSA are organized. Data flow is controlled by multiplexers according to the current operation going to be applied to input data. The HMAC module can either compute the hash value of the input data when ECDSA algorithm is executed or, it can compute the HMAC value of incoming data. The KDF depends on a shared secret value to start to generate the keys for E and HMAC. When data are signed, data to be hashed is taken from the output of the compressor. When digital signature is verified, data to be hashed is taken directly form the host (no compression is applied). For a encryption operation, data is encrypted by the E module, taking data from the compressor. In this case, HMAC computes the MAC value from the shared value. When data are decrypted, data are not compressed, so data coming from the host are processed by the encryption module and by the HMAC. KDF and HMAC are build around a core of the SHA-1 algorithm, which computes the hash value of a 512-bit data block.

Figure 5 shows the organization of the arithmetic units for both, elliptic points and large integers modulo n. All internal buses in both arithmetic units are m-bit wide. An Input/Output interface loads and reads new F_{2^m} values to/from the memories for the arithmetic units. The I/O interface does not can access protected information, like the private keys. Two memories are used for the elliptic curve arithmetic unit, one for storing the points involved in scalar multiplication and other for storing scalar values involved in the multiplication. The big arithmetic ALU only uses a memory for storing input and intermediate parameters. In this memory, the result of the HMAC module is stored for future read.

3.1. Data Compression

The compression module was designed using a systolic array approach. Its derivation was made by applying loop unrolling to the algorithm, taking the ideas given in [8]. The processing elements for the systolic array are depicted in figure 4.

The compression performance depends strongly on the size of two buffers in the LZ compression algorithm. An study of how compression ratio is affected by these sizes, and also more detail in the architecture of the compressor can be found in [12]. In the design of the compressor, the systolic array is composed of M type-I PEs and one Type-II PE. The latency of each codification step is in the worst case $(N + M)$.

3.2. Arithmetic units

Elliptic curve arithmetic unit executes either a scalar multiplication or an elliptic curve point's sum. Scalar multiplication is basically a sum of elliptic points, the operation kP is viewed as the sum of the point P with itself k times ($kP = P + P + ... + P$). This sum of points is one of two types: *Doubling* operation when two points are equal and *Add* operation when points are different. In this work, the binary method [6] in its left to right version is used. This algorithm allows to compute a *Doubling* and *Add* operations in parallel. Every sum of points requires several field operations, the number and type of them depends on the type of coordinates being used. In this work, the elliptic points are represented in affine coordinates and field elements are viewed as polynomials on the field $\{0, 1\}$. The *Doubling* operation requires the following field operations: 2 multi-

0-7695-2283-1/05 $20.00 © 2005 IEEE

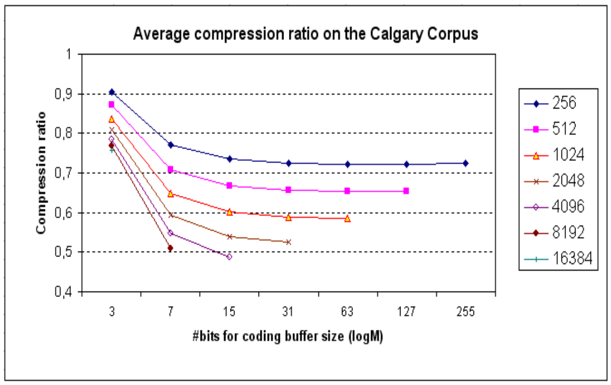

Figure 6. LZ compression performance for different buffers size

plications, 2 squaring, one inversion and 5 sums. The *Add* operation requires one inversion, two multiplications, one squaring and eight sums. A sum in the field F_{2^m} is a XOR operation and it is easily implemented in hardware. In this work, the multiplication is performed by a digit-serial multiplier, the squaring operation is performed with customized hardware and computed in only one clock cycle. The multiplier and squaring are based on the work reported in [11]. The inversion operation is carried out by direct division algorithm, described in [18].

Integer arithmetic, multiplication, sum, and modular reduction operations are performed according to algorithms reported in [9]. In these algorithms, modular reduction is performed by subtracting the module n until the result falls within the range [1, n-1]. The modular division operation required in ECDSA algorithm is performed as described in [18] for integer operands. Random numbers used in ECDSA and ECIES are generated by a m-bit random number generator implemented as a linear feedback shift register (LFSR) [19].

4. Implementation and Results

We synthesized and simulated the architecture for a Xilinx VirtexII XC2V6000-4ff1176 FPGA, using the ISE 6.x and Active-HDL 6.2 software tools. The compressor was synthesized for a searching buffer of size 1KByte and a coding buffer of size 15 bytes. These sizes were selected according to software results for the compression algorithm, testing different values for the buffers. Results of this test are shown in figure 6.

Arithmetic units were synthesized for the random curve recommended by NIST [13], for the finite field F_2^{163} using the irreducible polynomial $F(x) = x^{163} + x^8 + x^7 + x^3 + 1$. Synthesis results for each part of the full system is summarized in table 1.

Module	Slices	Utilization	BRAM
Compressor	9700	28%	0
HMAC	1339	3%	2
KDF	947	2%	2
E	19	1%	1
ECC-ALU	6080	17%	10
INT-ALU	2932	8%	5
RNG	177	1%	0

Table 1. Synthesis results

File	Size (bytes)	Time HW	Time SW	CR
progp	49379	94.54	265	0.47
obj1	21504	67.50	156	0.66
progc	39611	106.48	250	0.59
paper6	38105	111.78	250	0.61
paper5	11954	36.12	78	0.63
paper4	13286	42.28	109	0.65
paper3	46526	159.00	359	0.68
paper1	53161	163.81	421	0.63

Table 2. Timing results (ms) and compression ratio

The slower module was the elliptic curve ALU and it determines the clock frequency. On the contrary, the most area consuming module was the compressor. In this case most of the compressor area was occupied by the buffer that is implemented as a set of registers connected in cascade. According to the synthesis results, we post-simulate the compressor module, that determines the latency of data processing, emitting a codeword every 1039 cycles in the worst case. In table 2 the processing time for some files of the Calgary Corpus [21] is given. The system behavior was validated by comparing the simulation results in Active-HDL with those obtained with an equivalent software implementation, for data processing algorithms and for the arithmetic units. Modules as HMAC, SHA-1 and digital signature were verified using the test vector given in its specification.

Because other similar implementations have not been reported, we are not able to compare our results. However, we can compare the results of the elliptic curve arithmetic unit with some work that have been reported. A performance comparison of hardware implementations for scalar multiplication against each other is not straightforward because of different key size and FPGA technology used. In table 3, the scalar multiplication timing result we have obtained is compared with some hardware implementations mentioned earlier in this paper.

Reference	F_q	Platform	Time (ms)
This work	$F_{2^{163}}$	Xilinx XC2V6000	1.34
[14]	$F_{2^{163}}$	Altera EPIF10K250	80
[3]	$F_{2^{113}}$	Amtel AT94K40	10.9
[4]	$F_{2^{270}}$	Xilinx XC4085XLA	6.8

Table 3. Timing comparison for scalar multiplication

5. Conclusions and future work

We present a hardware architecture that combines lossless compression and public-key cryptography. The latency of the overall process is determined by the compressor, which can process up to 10 Mbps. As future work, the processing time for the compressor can be improved if the necessary time to look for a string is limited. For the cryptographic work, the time to perform the scalar multiplication can be improved if projective coordinates are used to represent the point of the curve and the Montgomery method is used to compute the scalar multiplication. The work and results presented in this paper seem to be the only hardware architecture that combines compression and encryption in a single chip as no work in the literature was found reporting on that. This hardware architecture for lossless compression and public key cryptography gathers the advantages of compression and cryptography, making the process of compression and encryption transparent to the final user.

References

[1] American Bankers Association. ANSI X9.62-1998: Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA).

[2] R. Dahab and J. López. An Overview of Elliptic Curve Cryptography. Technical Report, IC-00-10, http://citeseer.nj.nec.com/333066.html.

[3] M. Ernest *et al.* A Reconfigurable System on Chip Implementation for Elliptic Curve Cryptography over GF(2^n). In *Proc. of CHES'2002*, volume 2523 of *LNCS*, pages 381–399, Redwood Shores, CA, August 2002. Springer.

[4] M. Ernest *et al.* Rapid Prototyping for Hardware Accelerated Elliptic Curve Public Key Cryptosystems. In *Proc. of 12th IEEE Workshop on Rapid System Prototyping, RSP'2001*, pages 24–31, Monterey, CA, June 2001.

[5] R. Friend and R. Monsour. RFC 2395-IP Payload Compression Using LZS, available at rfc.sunsite.dk/rfc/rfc2395.html.

[6] D. Hankerson, L. López, and A. Menezes. Software Implementation of Elliptic Curve Cryptography over Binary Fields. In *Proc. of CHES'2000*, volume 1965 of *LNCS*, pages 1–24, Worcester, MA, August 2000. Springer.

[7] HiFn, Inc. The first book of compression and encryption, hifn whitepaper, www.hifn.com/docs/a/the-first-book-of-compression-and-encryption.pdf.

[8] S. Hwang and C. Wu. Unified VLSI Systolic Array Design for LZ Data Compression. *IEEE Transactions on Very Large Scale Integration Systems*, 9(4):489–499, August 2001.

[9] C. Kaya-Koc. RSA Hardware Implementation. Technical Report, TR-801, RSA Laboratories.

[10] N. Koblitz, S. Vastone, and A. Menezes. The State of Elliptic Curve Cryptography. *Designs, Codes and Cryptography*, 19(2/3):173–193, March 2000.

[11] J. Lutz. High Performance Elliptic Curve Cryptographic coprocessor. Master's thesis, University of Waterloo, 2003.

[12] M. Morales-Sandoval and C. Feregrino-Uribe. On the Hardware Design and Implementation of an FPGA-based Lossless Data Compressor. In *Proc. of ReconFig'04*, pages 22–26, Colima, Mex., September 2004.

[13] NIST. Recommended Elliptic Curves for Federal Government Use, http://csrc.nist.gov/csrc/fedstandards.html.

[14] S. Okada *et al.* Implementation of Elliptic Curve Cryptographic Coprocessor over GF(2^m) on a FPGA. In *Proc. of CHES'2000*, volume 1965 of *LNCS*, pages 25–40, Worcester, MA, August 2000. Springer.

[15] PKWare, Inc. Pkware Extends ZIP Standard to Support Strong Encryption, www.pkware.com/news_events/press_releases/2003/060203-appnote.html.

[16] Proginet, Co. Product Information Guide, www.proginetuk.co.uk.

[17] SEC1. Elliptic Curve Cryptography: Standards for Efficient Cryptography Group, http://www.secg.org.

[18] Shantz, S. C. From Euclid's GCD to Montgomery Multiplication to the Great Divide, Technical Report TR-2001-95, Sun Microsystems Laboratories, 2001.

[19] W. Stallings. *Cryptography and Network Security*. Prentice Hall, NJ, 2003.

[20] J. Storer and T. Syzmanski. Data Compression via Textual Substitution. *Journal of the ACM*, (29):928–951, June 1982.

[21] The Calgary Corpus (2001). http://links.uwaterloo.ca/calgary.corpus.html.

[22] J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23:337–343, May 1977.