

12-13-2010

Performance Analysis of Arithmetic Operations in Homomorphic Encryption

Jibang Liu

Electrical and Computer Engineering, Purdue University, liu285@purdue.edu

Yung-Hsiang Lu

Purdue University, yunglu@purdue.edu

Cheng-Kok Koh

Purdue University, chengkok@purdue.edu

Follow this and additional works at: <http://docs.lib.purdue.edu/ecetr>

Liu, Jibang; Lu, Yung-Hsiang; and Koh, Cheng-Kok, "Performance Analysis of Arithmetic Operations in Homomorphic Encryption" (2010). *ECE Technical Reports*. Paper 404.
<http://docs.lib.purdue.edu/ecetr/404>

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

Performance Analysis of Arithmetic Operations in Homomorphic Encryption

Jibang Liu

Yung-Hsiang Lu

Cheng-Kok Koh

TR-ECE-10-08

December 13, 2010

School of Electrical and Computer Engineering

1285 Electrical Engineering Building

Purdue University

West Lafayette, IN 47907-1285

Performance Analysis of Arithmetic Operations in Homomorphic Encryption

Jibang Liu, Yung-Hsiang Lu, and Cheng-Kok Koh

liu285@purdue.edu, yunglu@purdue.edu, and chengkok@purdue.edu

School of Electrical and Computer Engineering, Purdue University.

Homomorphic encryption allows computation on encrypted data and may protect privacy in cloud computing. Is homomorphic encryption ready for deployment? This paper is an attempt to answer the question.

As cloud computing is being adopted, protecting “outsourced” data becomes an essential issue. Several incidents highlight this concern. In May 2009, unintended sharing occurred on a site providing on-line office applications [7]. In October 2010, some applications in a social network violated the privacy policy [6]. As more personal (such as healthcare and tax report) and confidential (such as sales records) data are stored on-line, protecting privacy becomes an increasingly important topic. Many studies have been conducted to protect outsourced data. Among all solutions, *homomorphic encryption* [3] is a promising approach and Micciancio calls it the “Holy Grail” in Cryptography [5]. *Homomorphic encryption* [1] allows operations on encrypted data; thus, cloud servers may use encrypted data without access to the original data. Figure 1 shows the general framework. Homomorphic encryption is considered too expensive and remains an academic curiosity. This paper intends to shed some more light for understanding whether homomorphic encryption is ready for deployment and what directions should be pursued to make it practical.

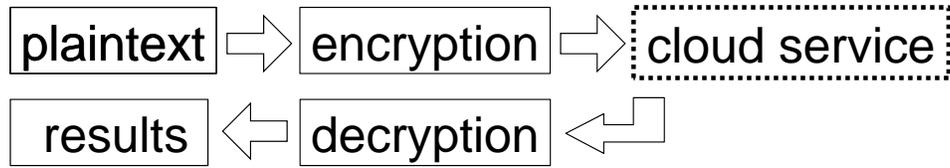


Figure 1: A general framework for using cloud service with data protection. Data are encrypted before being sent to the cloud server. The server performs computation on the encrypted data. The results are obtained by decrypting the data from the server. The solid lines represent the data owner; the dashed line means the server is not trusted.

Figure 2(a) and (b) illustrate the concept of homomorphic encryption. Suppose $x = \langle x_1, x_2, \dots, x_n \rangle$ is a sequence of n elements as the original unprotected data, also called *plaintext*. An operation f can be performed on x to obtain result $r = f(x) = \langle r_1, r_2, \dots, r_m \rangle$. Let $y = \langle y_1, y_2, \dots, y_n \rangle$ be the corresponding ciphertext; $y = \langle e(x_1), e(x_2), \dots, e(x_n) \rangle$ and e is the encryption operation. We can obtain x through decryption $x = d(y)$. We call the encryption and the operation homomorphic if $d(f(y)) = r$. In other words, the same operation can be applied to encrypted data and the result can be obtained after decryption, as illustrated in Figure 2(b). Homomorphic encryption is not another encryption algorithm (like AES or RSA). Instead, it is a property of *some* encryption algorithms; some encryption algorithms cannot be homomorphic, for example, if they are non-malleable [1]. The encryption algorithm illustrated in Figure 2 (b) is *deterministic*: for a plaintext x , a unique ciphertext y is created. Many encryption algorithms are *non-deterministic*: a plaintext x may be mapped to one of many possible ciphertexts. Non-deterministic encryption can provide better protection because it is difficult to know whether two different y 's correspond to the same x . This is illustrated in Figure 2 (c). The sidebar *Example of Homomorphic Encryption using Non-Deterministic Encryption* gives a numeric example. The sidebar also shows an example when $h(x) = x^3$ produces a wrong result for $x = \langle x_1 \rangle = \langle 2 \rangle$, illustrated by point a in Figure 2(d).

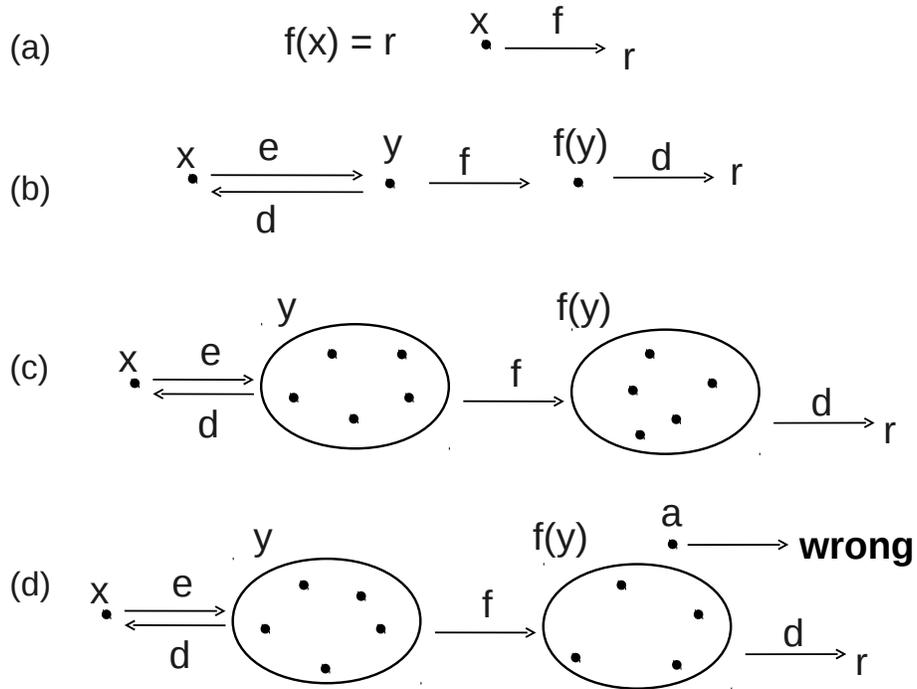


Figure 2: Overview of homomorphic encryption. (a) Operation f is performed on unprotected data x and produces result $r = f(x)$. (b) Homomorphic encryption means $y = e(x)$ and $d(f(y)) = d(f(e(x))) = r$. (c) Non-deterministic encryption maps one plaintext x to one of many possible ciphertexts. The circle is called *decryption radius* [4]. (d) In some cases, $f(y)$ may fall outside the decryption radius; thus, $d(f(y))$ is incorrect because it is different from $f(x)$.

The concept of homomorphic encryption was first proposed by Rivest, Adleman, and Dertouzos in 1978. For three decades, only a few operations (most noticeably addition and multiplication) are found homomorphic. The main problem is the introduction and accumulation of “noise” during computation performed on encrypted data. When the noise exceeds an acceptable range (called decryption radius), the decrypted result $d(f(y))$ is no longer correct. In 2009, Gentry proves that it is possible to construct a *fully* homomorphic encryption scheme for any operations [2, 3] by denoising frequently. The sidebar *Denoise Procedure* describes Gentry’s scheme. Even though it is theoretically possible to perform any operation on encrypted data, it is generally believed that the overhead is too high. A recent paper [4] provides a quantitative study about the overhead of denoising elementary symmetric polynomials of encrypted bits using Gentry’s algorithm. Our paper provides additional information about the overhead for performing arithmetic and relational operations. These operations form the foundation for future studies on computation using encrypted data. We create a library for these operations using the symmetric encryption scheme and denoising algorithm described in [3]. This paper quantitatively analyzes how execution time is affected by the types of operations, key sizes, data sizes, and the numbers of denoising.

The overhead of homomorphic encryption comes from four sources. (1) Encryption and decryption. (2) A small integer (for example, 32 or fewer bits) can become very large after encryption, depending on the size of the key. (3) Substantial overhead is introduced when arithmetic operations are implemented by software for encrypted data. (4) Denoising needs to be performed frequently to ensure the correctness of the intermediate and final results. The overhead of (1) and (2) has been widely studied. This paper focuses on (3) and (4).

We create a library for basic arithmetic and relational operations including “+_e”, “-_e”, “×_e”, “/_e”, “<_e”, and “==_e” on encrypted integers. The library is written in C++ with GNU Multiple Precision Arithmetic Library (GMP) 5.0.1 for handling large numbers. Our program runs on a server with 4 Intel Xeon L7555 1.87GHz processors and 128 GB memory. The GMP library and our program are compiled using the gcc version 4.1.2 with flag -O2 -m64. Neither our program nor the GMP library is parallelized. We use `gettimeofday` to measure the execution time taken by an operation.

Our implementation use the following steps. (1) A plaintext integer is first converted into the binary

representation. (2) Each bit is encrypted and then becomes a “big integer”, i.e. an `mpz_t` object in the GMP library. (3) The arithmetic operations are performed based on bit-wise operations using the algorithms described in [8]. The “ $<_e$ ” relational operation is implemented by subtracting the two operands and then taking the MSB of the result. The “ $=_e$ ” operation is implemented by subtracting two operands. If the two operands are equal, their difference is zero and every bit is zero. We detect whether every bit is zero by flipping each bit and multiplying them. If the result is one, the two numbers are equal; otherwise, the two numbers are different. These operations are implemented using the GMP library, such as `mpz_add` and `mpz_mul` on the encrypted bits. (4) Our method denoises the intermediate results if the accumulated noise exceeds a threshold. The procedure for determining the threshold is beyond the scope of this paper. We use `mpf_mul` from the GMP library to handle multiplication of floating-point numbers for denoising. (5) The ciphertexts are decrypted to get the final results. Figure 3 shows the example of implementing addition on two encrypted integers. The plaintexts are 3 and 2; each one is encrypted to three big integers (`mpz_t`). A full adder takes three big integers as the inputs and generates the sum and the carry; both are encrypted as `mpz_t` objects. The first full adder takes a plaintext 0 as the input carry. The last full adder discards the output carry. The four `mpz_t` objects corresponding to the sum bits are decrypted and the final result is 5.

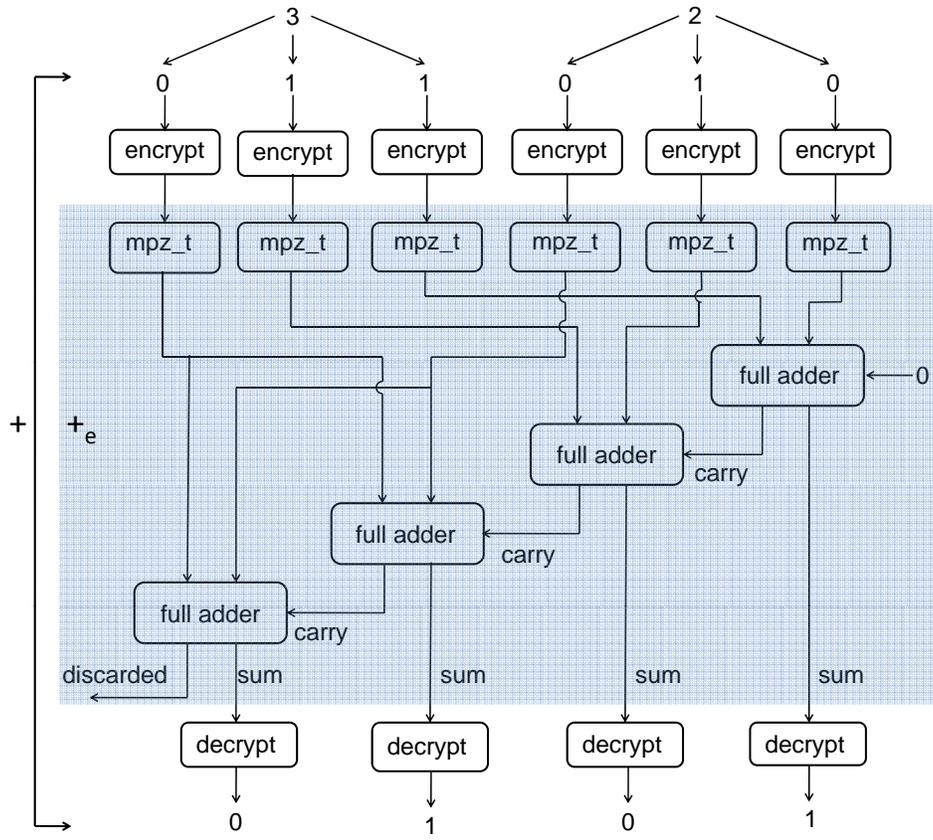


Figure 3: An example of implementing addition on two encrypted integers, 3 and 2. Each bit is encrypted to an `mpz_t` object. Both the input and output of a full adder are `mpz_t` objects. The shaded region represents the $+_e$ operation.

key size	generate key	encrypt	decrypt	mpz_add	mpz_mul	mpf_mul	denoise
1024 bits	12.35	8.39×10^{-3}	7.64×10^{-4}	4.97×10^{-5}	7.72×10^{-4}	0.45	774.3
2048 bits	43.86	9.46×10^{-3}	9.16×10^{-4}	6.73×10^{-5}	2.02×10^{-3}	2.43	3120.5
4096 bits	169.34	1.07×10^{-2}	1.25×10^{-3}	1.01×10^{-4}	5.73×10^{-3}	10.42	12384.1
8192 bits	594.36	1.21×10^{-2}	1.82×10^{-3}	1.67×10^{-4}	1.65×10^{-2}	53.13	53712.8

Table 1: Execution time for different operations on a single bit (unit: milliseconds).

Table 1 shows the time taken by different steps for operations on one bit with different key sizes. Key generation needs to be performed only once and the same key can be reused. The “encrypt” and “decrypt” columns show the time to encrypt and decrypt one bit. The “mpz_add” and “mpz_mul” columns show the time to perform an addition and a multiplication on two encrypted bits. The “mpf_mul” shows the time to perform the multiplication on one floating point number with one encrypted bit for denoising. The last column in this table shows the time to denoise once, using $|A| = 1024$ and $|B| = 15$ defined in the sidebar *Denoise Procedure*. As can be seen in this table, denoising takes much longer than the other operations. In actual computation, denoising may need to be performed multiple times for intermediate values.

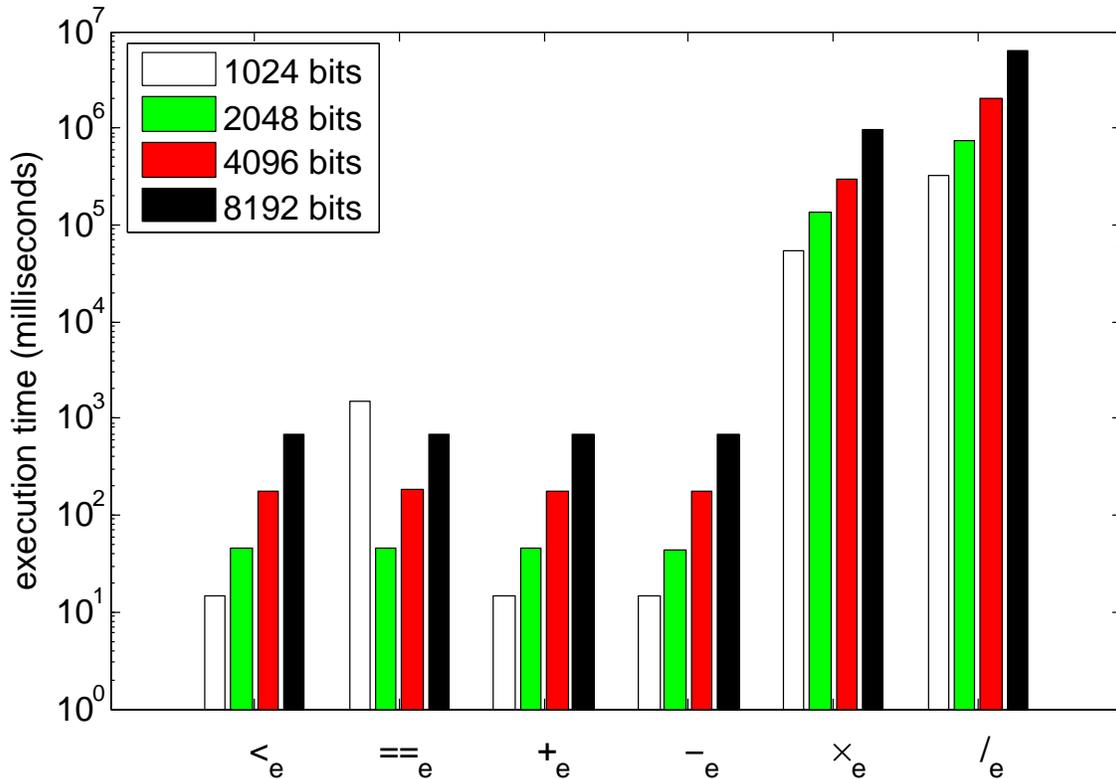


Figure 4: The time need to perform relational and arithmetic operations for different key sizes. Addition, subtraction, and less than are much faster because they do not require denoising. The plaintexts are two 8-bit integers.

Figure 4 shows the execution time to perform the six operations on two 8-bit integers with different key sizes. Three operations: “ $<_e$ ”, “ $+_e$ ” and, “ $-_e$ ”, are significantly faster because they do not need denoising. They take approximately 14.5 milliseconds for a key of 1024 bits and 675.0 milliseconds for a key of 8192 bits. The other operations require denoising and thus take longer time. For each operation, as the key size increases, the number of denoises decreases but the time for running one denoise increases. The “ $==_e$ ” operation needs denoising twice when the key size is 1024 bits and the execution time is 1.5 seconds. It does not use any denoise when the key size increases to 2048 bits and the execution time decreases to only 46.5 milliseconds. When the key size increases to 4096 bits and 8192 bits, it still does not need denoise; the execution time increases to 182.0 milliseconds and 691.2 milliseconds due to the longer keys. The “ \times_e ” and

“/_e” use more denoises, thus they take much longer time. The “×_e” operation uses from 70 to 17 denoises as the key size increases, and the time increases from 53.9 seconds to 962.3 seconds (16.0 minutes). The “/_e” operation uses from 415 to 117 denoises; its execution time increases from 323.4 seconds (5.4 minutes) to 6360.3 seconds (1.8 hours).

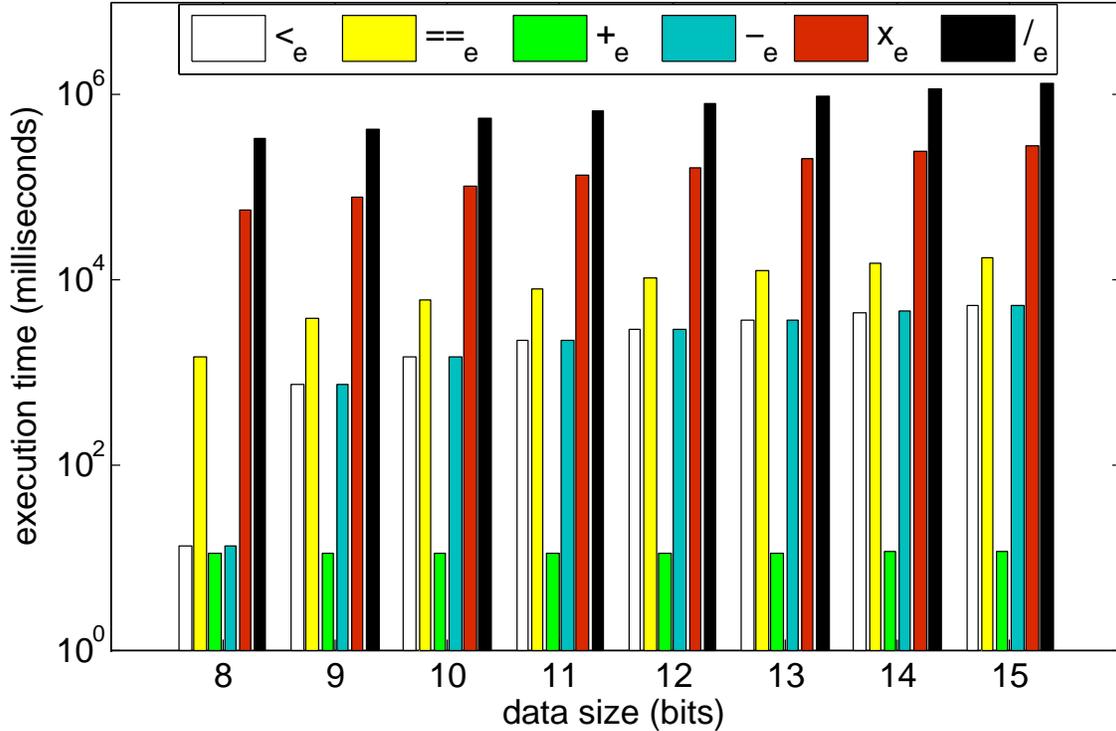


Figure 5: The time for performing six operations with different data size for the plaintexts. The encryption uses a key of 1024 bits.

Figure 5 shows the execution time to perform the six operations when the plaintexts increase from 8 to 15 bits. The encryption uses a key of 1024 bits. We do not consider data with more than 15 bits, because multiplication may produce the results exceeding a 32-bit integer. The execution time increases with data size because more bits are needed, more bit-wise operations are performed, and more denoises are used. When the plaintext data has 8 bits, the “ $<_e$ ” and “ $-_e$ ” do not need any denoise and take 14.1 milliseconds. They start to use denoise when data has 9 bits; one denoise is used and the execution time is 771.8 milliseconds. The number of denoises in “ $<_e$ ” and “ $-_e$ ” gradually increase to 7 when data has 15 bits, and the time increases to 5.2 seconds. When data size is 8-bit, “ \times_e ” and “ $/_e$ ” use 70 and 415 denoises, and take 55.7 seconds and 328.9 seconds (5.5 minutes). As data size increases to 15-bit, they use 350 and 1654 denoises, and take 278.5 seconds (4.6 minutes) and 1312.7 seconds (21.9 minutes). As shown in the figure, “ $+_e$ ” takes the shortest time. The execution times of “ $<_e$ ” and “ $-_e$ ” are close, because their

bit-wise operations are almost the same except the last step in " $<_e$ ", taking the MSB of the results, which does not introduce any noise. The " $<_e$ " and " $-_e$ " take longer time than " $+_e$ " because they perform extra two's-complement operations before using the " $+_e$ " operation. The " $==_e$ " first subtracts two operands and then multiply each bit of the result, therefore it takes more time than the " $-_e$ " operation.

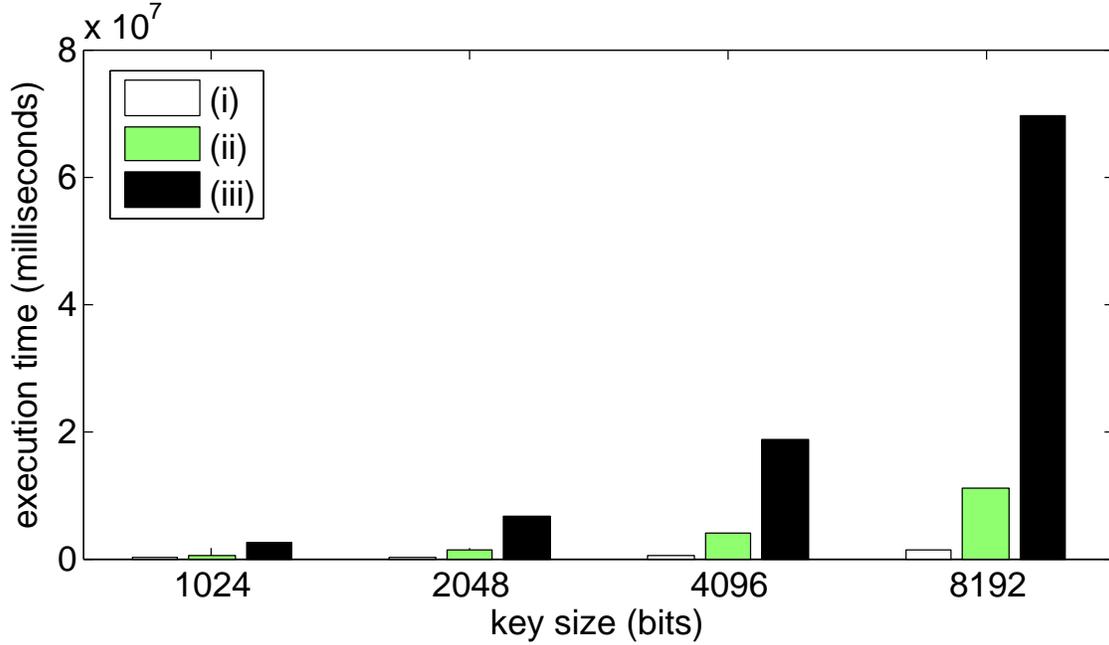


Figure 6: The time for computation with multiple operations.

We also consider computation with multiple operations: (i) $((3 \times 3) \times (3 \times 3)) \times 3 = 243$, (ii) $(13 \times (16 - 4)) \times ((-21/7) + 20)/9 = 156$, and (iii) $((13 + 16)^2 \times (4 - 21)^3) \times (21/5 + 20/(-9))^2 = -16527332$; they are all performed on encrypted data. Each plaintext integer uses as few bits as necessary and a sign bit is added as the MSB, for example, 13 uses five bits as 01101. Figure 6 shows the execution time of the three cases above with the key size increasing from 1024 bits to 8192 bits. The time rises for (i) from 85.2 seconds (1.4 minutes) to 1406.4 seconds (23.4 minutes) as the key size increases; denoising is performed from 100 to 26 times. For (ii), the execution time rises from 555.1 seconds (9.3 minutes) to 11028.1 seconds (3.1 hours); denoising is performed 703 to 205 times. For (iii), the execution time rises from 2466.7 seconds (41.1 minutes) to 69509.1 seconds (19.3 hours); denoising is performed from 3157 to 1261 times. Our evaluation shows that the overhead is mostly from denoising, which accounts for more than 99% of the total execution time in these cases. It can also be shown that as the key size increases, even though fewer denoises are required, each takes longer time and the total execution time is longer.

Even though computation on encrypted data is several orders of magnitude slower, we remain optimistic about the future of homomorphic encryption.

Computation on encrypted data is several orders of magnitude slower than computation on unprotected data. Hence, it is unlikely that cloud vendors would be able to offer services on encrypted data within a few years. However, we remain optimistic about the future of homomorphic encryption for the following reasons. (1) This paper presents *upper bounds* of the overhead because future studies can improve upon our current implementation. (2) Our implementation uses software in every step, from encryption to computation, from denoising to decryption. Some steps may use hardware accelerators, special-purpose co-processors or new instructions in future processors. Also, our implementation does not take advantage of multiple cores, available in most processors now. (3) We are confident that many efficient denoising algorithms will be developed in the next few years because protecting data is essential for cloud computing. (4) Research on homomorphic encryption is still at an early stage. Some government agencies have allocated funds to support this research; for example, in July 2010 US DARPA announced PROCEED (PROgramming Computation on EncryptEd Data). (5) It is possible to dramatically reduce the execution time if some operations do not need protection. For example, the average of two numbers $(a + b)/2$ can be performed on encrypted values for a and b but division by 2 may not need protection. In our implementation, this can be implemented by discarding the LSB and no noise is introduced. In this case, the cloud server knows division by two has been performed but the server does not know the values of a or b , nor their sum. (6) Many applications, for example, searching images or videos, do not require 100% accuracy. Existing search algorithms are imperfect even on unprotected data. It may be acceptable to lose a few more percentages of accuracy in exchange for better performance and protection of privacy.

References

- [1] Caroline Fontaine and Fabien Galand. A Survey of Homomorphic Encryption for Nonspecialists,. *EURASIP Journal on Information Security*, pages 1–15, January 2007.
- [2] Craig Gentry. Fully Homomorphic Encryption Using Ideal Lattices. In *ACM Symposium on Theory of Computing*, pages 169–178, 2009.
- [3] Craig Gentry. Computing Arbitrary Functions of Encrypted Data. *Communications of the ACM*, 53(3):97–105, March 2010.
- [4] Craig Gentry and Shai Halevi. Implementing Gentry’s Fully-Homomorphic Encryption Scheme. Technical report, IBM, 2010.
- [5] Daniele Micciancio. A First Glimpse of Cryptography’s Holy Grail. *Communications of the ACM*, 53(3):96–96, March 2010.
- [6] Emily Steel and Geoffray A Fowler. Facebook in Privacy Breach. *The Wall Street Journal*, October 18 2010.
- [7] Jessica E. Vascellaro. Google Discloses Privacy Glitch. *The Wall Street Journal*, March 8 2009.
- [8] John F. Wakerly. *Digital Design: Principles and Practices*. Prentice Hall, 3 edition, 1999.
- [9] Ping Zhu, Yanxiang He, and Guangli Xiang. Homomorphic Encryption Scheme of the Rational. In *International Conference on Wireless Communications, Networking and Mobile Computing*, pages 1–4, September 2006.

Example of Homomorphic Encryption using Non-Deterministic Encryption

Let p and q be two prime numbers and $n = pq$. The encryption function $e()$ is defined as [9]

$$y_i = e(x_i) = (x_i + pr_i) \pmod n,$$

here x_i is an integer smaller than p and r_i is an integer, chosen by the data owner. In this encryption, p and q are the keys and must be protected. The decryption function $d()$ is

$$x_i = d(y_i) = y_i \pmod n.$$

In real applications, p and q can be very large (hundreds or thousands of bits). In this example, we choose $p = 7$ and $q = 5$, $n = 35$. Let $x = \langle x_1, x_2 \rangle = \langle 2, 1 \rangle$ as the plaintexts. If we use 4 and 6 for r respectively in y_1 and y_2 , we can obtain

$$y_1 = (2 + 4 \times 7) \pmod{35} = 30$$

$$y_2 = (1 + 6 \times 7) \pmod{35} = 8.$$

Suppose the operation f is addition, i.e. $f(x) = x_1 + x_2$. We can obtain $r = 2 + 1 = 3$. If f uses the encrypted data, we can obtain $f(y) = y_1 + y_2 = 30 + 8 = 38$. Decrypting 38 and we can get

$$38 \pmod{7} = 3.$$

This is the same as r . Another example is multiplication. $g(x) = x_1 \times x_2 = 2 \times 1 = 2$. If multiplication is performed on y_1 and y_2 , we can obtain

$$30 \times 8 = 240 \text{ and } 240 \pmod{7} = 2.$$

This is the same as $x_1 \times x_2$.

Unfortunately, sometimes we may obtain incorrect results. Suppose $h(x) = x_1^3 = 2^3 = 8$. However, $h(y) = 30^3 = 27000$ and $27000 \pmod{7} = 1$; this is different from 8. This is an example when the result falls out of the decryption radius. Intuitively, $d(h(y)) \neq h(x)$ when $r \geq p$; the relation is actually more complex.

Denoise Procedure

Gentry presents a symmetric encryption scheme and the denoise procedure [3]. The encryption scheme uses an odd integer p as the key. Let $m_i \in \{0, 1\}$ be a one-bit plaintext. The corresponding ciphertext is $c_i = pq_i + 2r_i + m_i$, where q_i and r_i are integers chosen by the owner. Decryption is $d(c_i) = (c_i \bmod p) \bmod 2$, here $c_i \bmod p$ is the distance to the nearest multiple of p . This is also the noise associated to c_i . Decryption works only when the noise is less than p .

Each plaintext integer is first converted to its binary representation and arithmetic operations are performed on the bits. For example, a full adder has three input bits m_1, m_2 and m_3 and produces two output bits: $\text{MSB} = (m_1m_2 + m_1m_3 + m_2m_3) \bmod 2$ and $\text{LSB} = (m_1 + m_2 + m_3) \bmod 2$. After encryption of each input bit, the bit-wise operations are translated to additions or multiplications on the ciphertexts, i.e. $e(\text{MSB}) = (c_1c_2 + c_1c_3 + c_2c_3)$ and $e(\text{LSB}) = (c_1 + c_2 + c_3)$. This encryption is homomorphic because we can add or multiply (modulo 2) the bits by simply adding or multiplying their ciphertexts.

Decryption $d(c_i) = (c \bmod p) \bmod 2$ is equivalent to $\text{LSB}(c) \text{ XOR } (\lfloor c/p \rfloor)$, where LSB takes the least significant bit and $\lfloor \cdot \rfloor$ is the floor function. Getting LSB and computing XOR are easy; computing c/p is complicate and produces substantial amounts of noise. $A = \{1, 2, \dots, \alpha\}$ and B are two sets of integers. B is a proper subset of A : $B \subset A, B \neq A$. Suppose $\beta = |B|$. Since B is a proper subset of A , $\alpha > \beta$. $S = \{s_1, s_2, \dots, s_\alpha\}$ is an indicator set: $s_i = 1$, if $i \in B$, otherwise $s_i = 0$. Another set $T = \langle t_1, t_2, \dots, t_\alpha \rangle$ is created such that $\sum_{i \in B} t_i = 1/p$. Thus $\sum_{i=1}^{\alpha} s_i t_i = 1/p$. Each element in S is encrypted. Both S and T are sent to the cloud server for denoising. During denoising, the server computes $Z = \{z_1, z_2, \dots, z_\alpha\}$ with $z_i = cy_i \bmod 2$ ($i = 1, 2, \dots, \alpha$) by taking $\lceil \log_2(\beta) \rceil + 1$ bits after the binary point for each z_i . The denoised ciphertext is $\text{LSB}(c) \text{ XOR } \text{LSB}(\lfloor \sum_{i=1}^{\alpha} s_i z_i \rfloor)$. This is still encrypted but the ciphertext is pushed toward the center of the decryption radius. The server can detect the values of α and β . However, s_i is encrypted so the server does not know which elements in T are actually used. Brute force attacks require testing $\frac{\alpha!}{\beta!(\alpha-\beta)!}$ possible cases. This is computationally infeasible if $\alpha \gg \beta \gg 1$.