Applying Basic Encoding Rule (ITU-T X.690) on Integer Expressions using Arithmetic Circuit Homomorphic Encryption

Woon Hao Soh Kenneth
Cybersecurity & Digital Forensics
School of Informatics and IT
Temasek Polytechnic
Singapore
e-mail: kensohsg@gmail.com

Cheng Jing Kwa
Cybersecurity & Digital Forensics
School of Informatics and IT
Temasek Polytechnic
Singapore
e-mail: chengjing32@yahoo.com

Chern Nam Yap (MIEEE)
Cybersecurity & Digital Forensics
School of Informatics and IT
Temasek Polytechnic
Singapore
e-mail: cnyap@tp.edu.sg

***Abstract*—This is an extension of an ongoing research project on Arithmetic Circuit Homomorphic encryption (ACHE) which was implemented based on Fast Fully Homomorphic Encryption over the Torus (TFHE).**

This paper focuses on the integration of ASN.1 Basic Encoding Rule (ITU-T x.690) into ACHE with RFC7664. Tests were performed to calculate results of 3 operand integer expressions using additional, subtraction and multiplication circuits. Next, the paper will further discuss the protocol design and the method used to achieve the calculation of expressions, and the latency of the entire process.

Keywords-homomorphic encryption; arithmetic circuit; integer expressions; basic encoding rule

I. INTRODUCTION

A. Background

In terms of confidentiality, there are data at rest, data in transit and data in use. Homomorphic encryption tries to resolve issues concerning data in use [1]. As data is transmitted over the network before it is actually used, formatting and authenticating the data in transit is essential.

In earlier work, (RFC 7664) [2] dragonfly protocol had been implemented to provide authentication for data in transit. However, for any crypto protocol to work effectively, a standard interface description language is essential to define data structures within the protocol. In modern communication, the International Telecommunication Union Telecommunication Standardization Sector (ITU-T) [3] Abstract Syntax Notation One (ASN.1) is used.

This paper focuses on extending existing Arithmetic Circuit Homomorphic Encryption (ACHE) [4] with IETF RFC 7664 [2], with ITU-T X.690 BER [3] as well as providing integer expression computation beyond single operator operations through a combination of addition, subtraction and multiplication.

B. Purpose

This research aims to integrate ASN.1 Basic Encoding Rule [3] for the formal notation in the data transfer between nodes, with the eventual purpose of calculating the results of integer expressions, into ACHE [4]. Latency for various computations was also captured for further analysis.

The implementation consists of the following nodes/functions, namely, Sources (***Clients***), ***Cloud***, ***Output*** and ***Keygen*** (Key Generator). Figure 1 shows the architecture diagram of our set up.

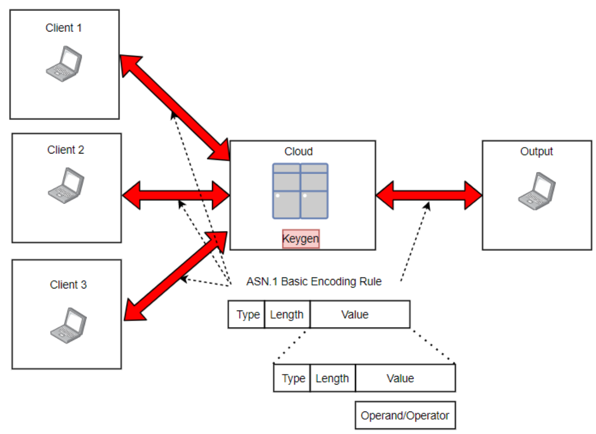

Figure 1. Basic encoding rule.

C. Scope

The computation is limited by both the availability of hardware and the software library used. A total of 6 Virtual Machines are used, each on Ubuntu 20.04 LTS, with Intel i7 single-core and 1Gibyte RAM.

In terms of software, computation is limited to 32-bits processing due to the limits of ACHE [4]/TFHE [5] at the current state.

II. APPLYING BER X.690

A. Overview

Abstract Syntax Notation One (ASN.1) is a standard interface description language by ITU-T, used to serialise and deserialise data structures across platforms. Basic encoding Rule (BER) X.690 [3], a set of ASN.1 encoding rules, allows for clear data transfer across nodes.

B. Integration and Purpose of ASN.1 BER

ASN.1 BER [3] is a widely recognised data transmission standard. By integrating BER with our research, future researchers can easily continue to make improvements to our work. Besides BER allows for efficient data transmission as it uses the Type-Length-Value (TLV) encoding scheme which enables data of multiple types and values to be transmitted in a single attempt. In this research, most encrypted data transmitted across nodes are encoded in BER. Table 1 provides a list of data types and values encoded in BER.

Type	Value	Use Case
Sequence	IP Address Operations Postfix	User Input sent from ***Output*** to ***Cloud***
Octet String	IP Address	***Client*** IP Address
Octet String	Operation	Operation code for Operator
Octet String	Postfix	Postfix Expression
IA5String	dataMac	MAC Address for Dragonfly Key Exchange
Octet String	dataKey	Private/Private Key
IA5String	dataScalarElement	Scalar/Element for Dragonfly Key Exchange
IA5String	dataStaAp	STA token for Dragonfly Key Exchange
Integer	dataFsize	Size of ciphertext value between ***Cloud*** and ***Client***
Octet String	dataContent	Ciphertext value between ***Cloud*** and ***Client***
IA5String	dataIndicator	Indicator of successful transmission of ciphertext value between ***Cloud*** and ***Client***
Integer	dataAnsSize	Size of computed ciphertext answer
Octet String	dataAnswer	Computed ciphertext answer

Table 1 Data Dictionary for the BER Frame Transmission

III. INTEGER EXPRESSION CALCULATION

A. Dragonfly Key Exchange (RFC 7664) [2] Authentication

Dragonfly Key Exchange is used to authenticate nodes participating in the integer expression calculation. At the start of each calculation cycle, nodes will perform the dragonfly key exchange with each other to generate a pair of homomorphic <u>private/public</u> keys, which are used to encrypt source numbers used in the calculation. The size of the <u>private/public</u> keys generated is approximate ~78.25MByte each. For the purpose of this research, port 4380 is used in all key exchanges with the Key Generator node. Figure 2 below is a message sequence chart for the Dragonfly Key Exchange.

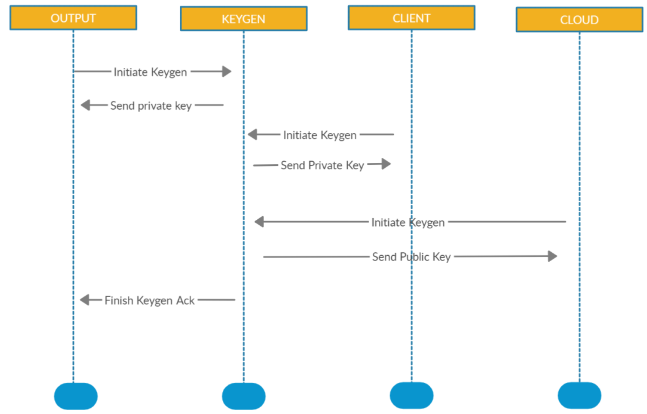

Figure 2. RFC 7664 Message Sequence Chart [6].

B. User Input Handling

The ***Output*** node accepts user input and displays of the final results. User inputs are validated using regular expression (regex) and generally the information is integer expression and corresponding IP addresses where data is located. User input is accepted in the form of alphabet letters, for operands, and mathematical operators, for example: 'A+B', where 'A' and 'B' are operands and '+' is an operator. If any part of the user input is considered invalid, the user will be asked to re-enter a valid input.

The ***Output*** node then converts the valid user input from infix notation to postfix notation, then filters operands from operators.

Postfix notation [7] refers to a format where operators are specified after operands. Figure 3 provides a simple example of the difference between postfix notation and infix notation. The use of postfix notation makes it easy to determine the order of which operands and operator should be considered first for computation, based on operator precedence.

The arithmetic expression **A + B * C** consists of **operands A, B, C** and **operators *, +**

<u>Postfix</u>
Format where operator is specified **after** the operands
A B C * +

<u>Infix</u>
Format where operator is specified **between** the operands
A + B * C

Figure 3. Postfix notation.

Output node measures the number of operands and requests, from the user, IP addresses which correspond to the ***Client*** nodes. Each IP address that is entered will be sent an "ICMP echo request" [8] twice to ensure that the host on that IP address is available. If "ICMP echo response" failed, the user will be prompted to re-enter a valid IP address. ***Output*** node also converts mathematical symbols for operators into operation codes, where addition (+) is referred to as '1', subtraction (-) as '2' and multiplication (*) as '3'.

With IP addresses and operation codes, ***Output*** node performs a one-time Dragonfly key exchange with ***Cloud*** node to obtain a shared key that will be used to encrypt the user input. The encrypted user input, which consists of IP addresses, operation codes and the postfix expression, is then sent via BER to the ***Cloud*** node on port 4381 [9].

C. Integer Expression Computation

The ***Cloud*** node receives on port 4381, decrypts the encrypted user input sent from ***Output*** node. At this stage, the ***Cloud*** node possesses 3 types of data: *IP Addresses*, *Operations Codes* and the *Postfix Expression* obtained earlier by ***Output*** node. ***Cloud*** node iterates through the *Postfix Expression*, distinguishing characters by alphabetical letters and operators. *IP Addresses* and *Operations Codes* are appended into 2 separate lists, ipList and opList respectively while maintaining its sequential order as provided by the *Postfix Expression*. To perform calculations of expressions with 3 operands, the ***Cloud*** node reads the last 2 IP addresses from the ipList and first operation code from the opList. ***Cloud*** node reads each *IP Addresses* and opens a socket connection on port 4381 to the ***Client*** to request for a source data “source buffer”, which was then encrypted using *Homomorphic Private Key*.

Based on the operation code read from the ciphertext, ***Cloud*** node performs addition, subtraction or multiplication on the ciphertext values request from the clients and write the computed value to a “result buffer”.

Before continuing with the second half of the expression, ***Cloud*** node read the following IP addresses and operation code and requests a third cipher text value. The ***Cloud*** node then copies contents from the ‘result buffer’, which consists of the answer from the first part, to append to the ‘source buffer’ file. Similar to the first part, ***Cloud*** node performs addition, subtraction or multiplication on the ciphertext values in the “source buffer” and overwrites the final answer to “result buffer”. The contents of the “result buffer” is read and sent back to ***Output*** node on port 4381 for decryption and display.

The entire process is summarised in Figure 4 below.

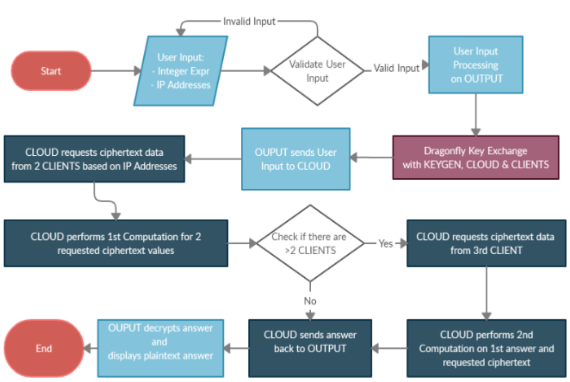

Figure 4. Steps of integer expression computation.

Figure 5 shows the message sequence chart for the integer expression computation process.

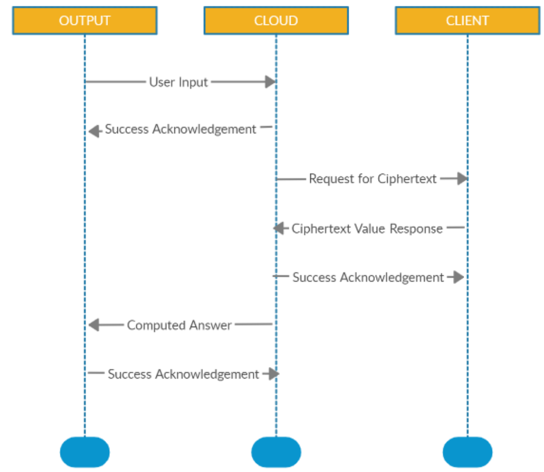

Figure 5. Integer expression computation message sequence chart [6].

D. Artihmetic Operations

Integer expression calculation is limited to addition, subtraction and multiplication. Based on previous research, each of these arithmetic operations was implemented separately [4]. For this research, the separate files for these 3 operations were combined into 1 single code so as to allow dynamism and flexibility in choosing operators for expression calculation.

The addition operation is based on full adder logic (1) depicted as follows and implemented on 32-bit (signed long) integer. This results in a range of values between -2,147,483,647 and + 2,147,483,647 [10].

$$\begin{aligned} \text{SUM} &= (\text{B} \oplus \text{Cin}) \oplus \text{A} \\ \text{Cout} &= \text{Cin} \oplus [\,(\text{A} \oplus \text{Cin}) \,.\, (\text{B} \oplus \text{Cin})\,] \end{aligned} \tag{1}$$

The subtraction operation is implemented using a Two’s complement number system (2). Similar to addition, it is implemented on 32-bit (signed long) integer. For two’s complement number system, the value ω of an N-bit integer $a_{N-1}\, a_{N-2} \ldots a_0$ is shown in Equation 2. Using N bits, all integers from – (2^{N-1}) to $2^{N-1} - 1$ can be represented.

$$\omega = -a_{N-1}2^{N-1} + \sum_{i=0}^{N-2} ai2i \tag{2}$$

The implementation of multiplication requires a bit shifter and adder operations. Multiplication in binary works by performing the “AND” operation on the least significant bit (LSB; right-most bit) of the multiplier to a partial product and repeats until the multiplier completes the “AND” operation on the most significant bit (MSB; left-most bit). Finally, each partial product is added using the addition operation. Multiplication is also implemented on 32-bit (signed long) integer.

Figure 6 provides an example on how binary multiplication works:

				1	0	1	0_2	(10_{10})	[Multiplicand]
*				1	0	1	1_2	(11_{10})	[Multiplier]
AND				1	0	1	0		[Partial Product]
			1	0	1	0			
		0	0	0	0				
	1	0	1	0					
Adder	1	1	0	1	1	1	0		[Final Product]

Figure 6. Multiplication.

E. *Latency*

For this research, all machines are connected via virtual interface – Intel Pro 1000, i.e. 1 Gbit/s connections.

The latency of the entire process was split into 4 segments: User Input Processing, Data Request, Key Generation and Integer Expression Calculation. The average timing for User Input Processing and Data Request is 6.90 seconds and 15.4 seconds, respectively. Latency for 6 unique 3-integer expressions was also measured with varying types of operations performed. Figure 7 shows the mean number of seconds, the ***Cloud*** node took to calculate each integer expression.

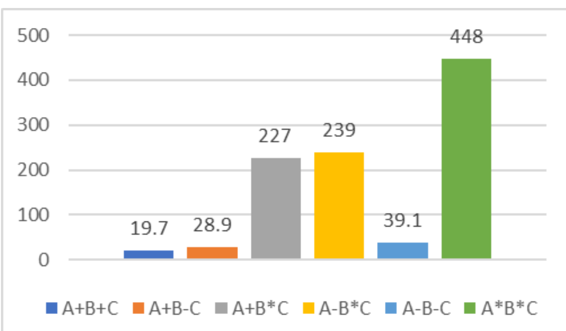

Figure 7. Homomorphic encyption compution latency.

The average overall timing of each process in seconds is seen in Figure 8. This includes the timing for pre-optimised Dragonfly Key Exchange.

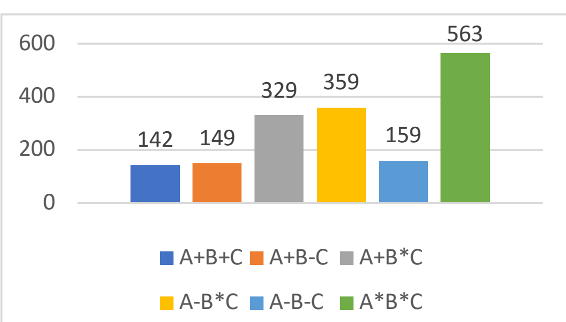

Figure 8. Overall computation latency.

IV. Optimisation

For this research, the Dragonfly Key Exchange process was optimised through simplification of the earlier process [2] and through multi-threading.

Instead of re-generating the same Homomorphic Encryption private key (secret.key) each time the ***Keygen*** node conducts Dragonfly Key Exchange with the ***Output*** or ***Client*** node, the ***Keygen*** node generates the secret.key only once then copies the same key to the ***Output*** node and each ***Client*** node simultaneously with multi-threading. As seen in Figure 9, this reduces the latency for the Dragonfly Key Exchange process by a mean of 22 seconds (to 3 significant figures), which is approximately 23%.

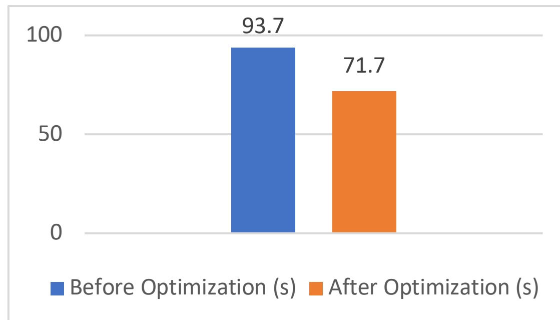

Figure 9. RFC 7664 optimisation.

Latency difference was also measured for Dragonfly Key Exchange involving 1, 2 and 3 unique clients. The difference in latency for each additional client is approximately 4.7 seconds (to 3 significant figures). This is shown in Figure 10 below:

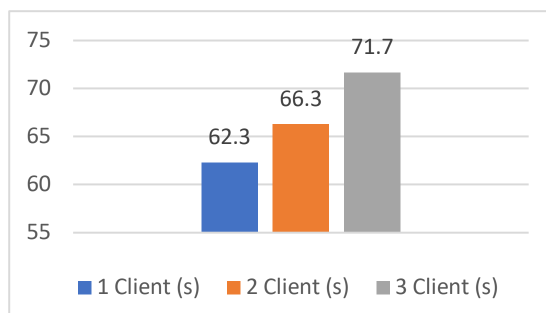

Figure 10. Latency between having varous number of Clients.

Optimisation in the Dragonfly Key Exchange reduces the latency for the entire process, while ensuring that the security aspect of key generation and exchange is not compromised.

V. Limitations

The current system is unable to reliably compute expressions involving divisions. This is because the data type used is an integer instead of floating-point. In addition, the answer size processed by the cloud must not exceed 32-bit (signed long) integer, or an integer overflow will occur.

VI. CONCLUSION

Generally, using ITU-T X.690 BER provide opportunity to process integer expressions involving addition, subtraction and multiplication to 3 different operands.

Some optimisation were made, however there may be further possibilities.

ACKNOWLEDGMENT

This research was made possible with the support of researchers of the previous Arithmetic Circuit Homomorphic Encryption (ACHE) projects. Gratitude is also extended to developers of TFHE [3] for code available on GitHub.

REFERENCES

[1] F. Armknecht, C. Boyd, K. Gjøsteen, A. Jäschke, C. Reuter and M. Strand, "A Guide to Fully Homomorphic Encryption," Cryptology ePrint Archive, 2015.

[2] D. Harkins, "Dragonfly Key Exchange. RFC 7664. (Nov. 2015).," 2015.

[3] X. ITU-T, "ITU-T Recommendation X.690 (2015) | ISO/IEC 8825-1:2015, Information Technology - ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER).," ITU-T, 2015.

[4] J. S. R. W. a. C. N. Y. R. Kee, "Arithmetic Circuit Homomorphic Encryption and Multiprocessing Enhancements," in *2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, Oxford, 2019.

[5] I. Chillotti, "Multi-Key Homomophic Encryption from TFHE," [Online]. Available: https://github.com/ilachill/MK-TFHE. [Accessed Oct 2019].

[6] Z. ITU-T, "ITU Recommendation Z.120 message sequence chart (MSC)," ITU, 2011.

[7] D. M. Kasprzyk, C. G. Drury and W. F. Bialas, "Human behaviour and performance in calculator use with Algebraic and Reverse Polish Notation," Ergonomics, 22 (9): 1011–1019, 1979.

[8] J. Postel, "Internet Control Message Protocol," IETF, Sep 1981.

[9] L. E. J. T. M. W. S. C. M. Cotton, "Internet Assigned Numbers Authority (IANA) Procedures for the Management," Internet Engineering Task Force, Aug 2011.

[10] I. 9899:2018, "Information technology — Programming languages — C," ISO/IEC, 2018.

[11] J. O. S. III, Mathematics of the Discrete Fourier Transform (DFT), with Audio Applications, W3K Publishing, 2007.