View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Research Commons@Waikato

Privacy Preserving Computation by Fragmenting Individual Bits and Distributing Gates

Mark A. Will
Cyber Security Lab
The University of Waikato
Hamilton, New Zealand
Email: willm@waikato.ac.nz

Ryan K. L. Ko
Cyber Security Lab
The University of Waikato
Hamilton, New Zealand
Email: ryan@waikato.ac.nz

Ian H. Witten
Department of Computer Science
The University of Waikato
Hamilton, New Zealand
Email: ihw@waikato.ac.nz

***Abstract*—Solutions that allow the computation of arbitrary operations over data securely in the cloud are currently impractical. The holy grail of cryptography, fully homomorphic encryption, still requires minutes to compute a single operation. In order to provide a practical solution, this paper proposes taking a different approach to the problem of securely processing data. FRagmenting Individual Bits (FRIBs), a scheme which preserves user privacy by distributing bit fragments across many locations, is presented. Privacy is maintained by each server only receiving a small portion of the actual data, and solving for the rest results in a vast number of possibilities. Functions are defined with NAND logic gates, and are computed quickly as the performance overhead is shifted from computation to network latency. This paper details our proof of concept addition algorithm which took 346ms to add two 32-bit values - paving the way towards further improvements to get computations completed under 100ms.**

***Index Terms*—homomorphic encoding; secure processing; encryption; distribution; data privacy; cloud computing;**

I. Introduction

The necessity to process data securely has been been discussed for decades [1], and has become even more crucial with our dependence on the cloud [2]. Services such as Dropbox [3] encrypt users data and manage the keys themselves. But malicious hackers may break the key management system, and rogue employees of the cloud service can easily view the data [4]. Dropbox was actually hosted on another third-party cloud service provider [5], meaning two sets of employees have potential access to user data.

On the other hand, privacy-focused storage companies such as Mega Limited [6] ask end users to encrypt all data themselves before uploading it to the cloud, rendering it unreadable. This guarantees strong security and privacy, but means that the cloud service cannot provide much functionality. In order to combine the features of Dropbox and Mega, we need to be able to protect data, while retaining the ability to compute over it. Fully homomorphic encryption [7] has been the proposed solution to this problem for years now, however is yet to be made practical. Performance figures in Section II show that it still takes minutes for a single operation.

The FRagmenting Individual Bits (FRIBs) scheme, has been designed to distribute each individual bit across many service providers, while still allowing Negative-*AND* (*NAND*) operations to be computed. We likened our proposed idea to the New Zealand terminology of ‘fribs’, which are small pieces of unwanted wool removed after shearing. If we say a “bit” is the woollen fleece, then it cannot be recreated without all the fribs and wool. Distributing the bit fragments can be seen as exporting the fribs and wool to different locations, known as fragment servers. Once exported, the bit fragments can be processed securely, by building functions from *NAND* gates.

This is further described in Sections III, IV and V. The implementation and practicality of FRIBs is presented in Section VI, and shows a performance increase over fully homomorphic encryption. FRIBs protects users data and privacy because each fragment server only gets a small portion of the data. We further evaluate the security provided by FRIBs in Section VII, and provide examples for additional security in Section VIII. To show that FRIBs can be applied to many different applications, we discuss a few use cases in Section IX, comparing to the current state-of-the-art solution.

II. Related Work

A. Homomorphic Encryption

Homomorphic encryption exists in 2 flavours: Partially Homomorphic Encryption (PHE) and Fully Homomorphic Encryption (FHE). PHE supports a single operation, for example, addition or multiplication. Where FHE can support many operations computed over encrypted data.

Cryptographic schemes supporting single homomorphic operations have been around since RSA was proposed in 1978 [8]. For some applications, only one operation is required, and in these cases PHE is an ideal solution [9][10][11][12]. However many applications need multiple operations, and must therefore use FHE.

FHE was only proven plausible by Gentry as late as 2009 [13], many years after PHE. Wang *et. al.* [14] showed performance results of a revised FHE scheme by Gentry and Halevi [15] in 2015 for the recrypt function. CPU and GPU implementations took 17.8 seconds and 1.32 seconds respectively, using a small dimension size of 2048 [14]. A medium dimension size of 8192 took 96.3 seconds and 8.4 seconds for the same function [14].

Currently hardware implementations [16] of FHE schemes cannot give practical processing times, so it will be difficult to make this technology usable in the real world. Combined

with the fact that quantum computing is making huge advancements [17][18], having data protected by traditional encryption schemes (for example RSA [8], Diffie-Hellman [19] and elliptic curves [20]) may not be as feasible in the future as it is today. Lattice-based encryption [13] could be a solution; however, it will result in even larger key sizes than current impractical FHE schemes.

B. Hardware Solutions

The state-of-the-art secure processor, AEGIS [21], was designed to only reveal the data inside the processor. Therefore any data leaving the processor is encrypted. This protects against a range of software and physical attacks. But AEGIS still has security vulnerabilities in the form of side-channel-attacks [22][23]. This attack vector analyses information "leaked" from the physical execution of a program, for example power consumption [24] or electromagnetic radiation [25]. Other limitations of secure processors are the practicality of deployment in the cloud. By definition, the cloud should be flexible and adaptive, often viewed as abstracting services from products [2], but by creating services reliant on custom hardware, we lose the core essence of what the cloud should be.

C. Distribution

Cloud providers distribute their services for features like lower latency, load-balancing, and redundancy [26][27]. Distribution can also provide better security and data protection by distributing user data over many servers, for example splitting database columns [28]. Then if a server is compromised, only some of the data is lost. Some PHE schemes have threshold variants which allow decryption to be split across many servers [29], and have primarily been used for voting schemes [30][9][12]. This provides extra protection to the decryption key, as each server only possesses a part of it. This is similar to encrypting data, as the time required to break the encryption can be compared to time required to break into all servers. Therefore distribution can enhance security, not just give better performance, which is the traditional school of thought.

D. Homomorphic Encoding

Encoding in general has been homomorphic for a while, audio and video being only a few examples. However it has not been readily explored for processing data securely. In 2015 we proposed Bin Encoding [31], a lossy encoding technique for securely searching strings. This showed that encoding can process data in a secure manner, while providing additional functionality over the likes of homomorphic encryption. It can also take advantage of the security given by distributing the data across many service providers.

III. Problem Formulation

Given how impractical FHE is currently, we have taken a different approach to the concept of processing data securely. Following on from our previous work [31], we propose encoding and distributing data to different cloud service providers.

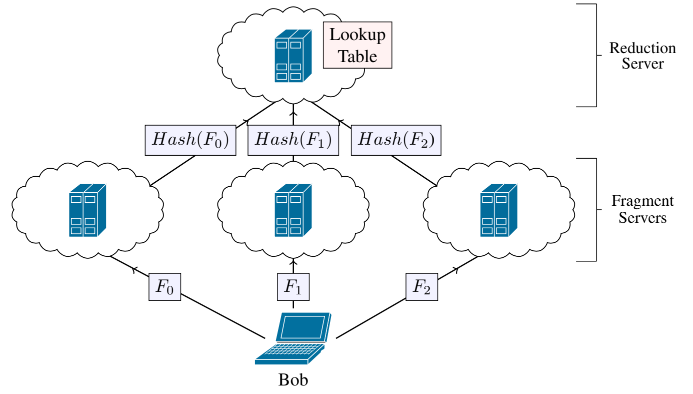

Fig. 1. System Model with 3 Fragment Servers and 1 Reduction Server.

A. System Model

Our scheme is aimed at all cloud users, personal and enterprise. Figure 1 shows a typical use case for a personal user. Bob is recording his health data (heart beat or sleep patterns) collected from a activity tracker. He wants to store this data in the cloud, but does not want his data shared with third parties. Therefore Bob needs his data protected, but still wants the ability to compute over the data, to find his average sleep durations for example.

As the data is being uploaded, each bit is split into fragments: F_0, F_1 and F_2 in Figure 1. Each fragment is protected using public-key encryption and sent to a seperate fragment server managed by different cloud providers. When these fragments are stored on the servers, they are encrypted using a user unique key (managed by the cloud). Each fragment server has less than $1/3$ of the data (explained further in Section IV), therefore Bob's data is kept private.

Fragment servers can perform *NAND* operations, allowing for arbitrary algorithms to be computed over Bob's health data. As operations are computed, the fragments will grow in size. Then at a pre-determined stage, these fragments must be reduced in size. The fragment servers apply a hash function with a user defined salt value. The hash values are sent to a reduction server, which has a lookup table previously uploaded by Bob, as shown in Figure 1. The lookup table contains encrypted fragments for each fragment server. Once a fragment is reduced, more operations can be computed. Then Bob can be securely sent all the result fragments to decode the result value.

B. Design Goals

The proposed scheme meets the following design goals.

- **Support for all operations.** We designed our scheme to support many operations, including conditional functions, by implementing a *NAND* gate.
- **No single server can reveal the full data.** To protect privacy, each server should not be able to decode any value (for example a 32-bit integer).
- **Full cloud service.** The scheme should be easy to implement on current cloud infrastructure, for example

Amazon AWS [32] or Microsoft Azure [33], and not require any special hardware or equipment.
- **Practical Performance.** Our scheme should be usable, allowing todays users of the cloud to be protected, while still getting computational functionality.

C. Threat Model

The threat model against which we evaluate our method is based on the following assumptions: 1) the communication channel between each fragment server, the reduction server, and the client is secure; 2) each server encrypts user data before storing it to disk; 3) each fragment server has no knowledge on other fragment servers, and 4) data is stored across multiple cloud service providers.

Based on these assumptions, there are two types of attacker to evaluate FRIBs against: a malicious insider, and a malicious user/outsider. Both present similar threats, however a malicious insider has an advantage because they already have access to one cloud service providers system. If a malicious insider manages to bypass all internal security, for example access polices and permissions, then they can discover $1/(2N)$ of the data. Now they become the same as any other malicious user, as they can try to break into all the other cloud service providers. The other option is to try and break the single set of fragments. This summaries into two attack vectors: breaking the data with one set of fragments, and getting all the fragments from each system.

TABLE I
SIMPLE *AND* FRAGMENTATION.

Value	F_0	F_1
low	0	0
low	1	0
low	0	1
$high$	1	1

TABLE II
FRAGMENTATION WITH MANY STATES.

Value	F_0	F_1
low	0	0
low	1	0
low	0	1
low	2	0
low	0	2
low	2	2
$high$	1	1
$high$	2	1
$high$	1	2

TABLE III
FRAGMENTATION WITH ONLY ONE SERVER RECEIVING 0.

Value	F_0	F_1	F_2	F_3	F_4
low	0	1	1	1	1
low	1	0	1	1	1
low	1	1	0	1	1
low	1	1	1	0	1
low	1	1	1	1	0
$high$	1	1	1	1	1

IV. FRAGMENTING INDIVIDUAL BITS

In the proposed scheme, data privacy is achieved by fragmenting individual bits, where the fragments are spread across many service providers and locations. Only when the fragments are combined, can the bit value be decoded. This follows the same principle of a Threshold cryptosystem [29], which has N entities, but only requires t entities to decrypt a value (where $t < N$). Therefore if t entities are compromised, then the encrypted data is no longer protected.

Given a value $\{0, 1\}$ or $\{$*low, high*$\}$, the *AND* function is used to encode/fragment the bit (why the *AND* function is used instead of *NAND* is explained in Section V-B). An example is shown in Table I where a value is encoded into two fragments. A potential problem with this example is that 50% of the fragments are 0. Assuming an equal probability $(50 : 50)$ between high and low bits before encoding, each servers can solve $\approx 1/3$ of the bit values (using the fact that $1/2$ are low, and $2/3$s of the fragments are 0 for low values). Depending on requirements this could be seen as too much information leakage, even though complete values (32-bit integers or 8-bit characters for example) are still unknown.

One technique to reduce the number of 0 fragments is to introduce more fragment states. Table II gives an example of three states for two servers, resulting in $1/3$ of the fragments equalling 0. The fragmentation used in Table II is $F_0 \wedge F_1$, where the value 2 is *low* unless the other fragment is *high*. Now each server can only solve $\approx 1/4$ of the bit values. However the easier solution is to increase the number of fragments, and only allow one fragment to be 0 for an encoding, as shown in Table III. This results in only $1/(N{+}1)$ fragments equalling 0, where N is the number of fragments, and a server only having knowledge of $1/(2N)$ of the bit values.

V. DISTRIBUTED NAND GATE

A. Operation

Now that the bits have been fragmented, we need to be able to compute basic operations over them such as addition and multiplication. Fragmenting values for either an addition or multiplication is trivial. For example if we have two numbers, 12 and 10, we can fragment them into $(6, 6)$ and $(8, 2)$ respectively. Adding the fragments together gives $(14, 8)$, which when joined together results in 22. Multiplication is similar, with the fragments $(3, 4)$ and $(2, 5)$, multiplication gives $(6, 20)$, outputting 120. The challenge is being able to compute both operations on the same set of data.

In order to compute many types of operations while only implementing a single function, a universal logic gate is used. FRIBs implements a *NAND* gate, as they are preferred over *NOR* gates in electrical applications [34]. Unlike the addition or multiplication example, the result of a *NAND* function is dependant on the other fragments. Equation 1 compares the Exclusive-*OR* (*XOR*) and *NAND* functions, where the *NAND* function gives the wrong result. Therefore FRIBs maintains state so that when joining the fragments together, the correct result is given.

$$
\begin{array}{c|ccc|c}
 & F_0 & & F_1 & \text{Result} \\
\hline
A & 0 & \oplus & 0 & 0 \\
 & \oplus & & \oplus & \oplus \\
B & 1 & \oplus & 1 & 0 \\
\hline
 & 1 & \oplus & 1 & 0
\end{array}
\qquad
\begin{array}{c|ccc|c}
 & F_0 & & F_1 & \text{Result} \\
\hline
A & 0 & \bar{\wedge} & 0 & 1 \\
 & \bar{\wedge} & & \bar{\wedge} & \bar{\wedge} \\
B & 1 & \bar{\wedge} & 1 & 0 \\
\hline
 & 1 & \bar{\wedge} & 1 & \textbf{?}
\end{array}
\tag{1}
$$

B. Maintaining State

To keep the states of each operation we use a simple technique of concatenating each fragment together to be computed/reduced later. The first operation with both fragments of 1, will be concatenated to 11. If we continued concatenating we would lose the order of *NAND* operations, as demonstrated in Equations 2 and 3 where the same fragment values give different results.

$$\begin{array}{c|rcr|c} & F_0 & & F_1 & \text{Result} \\ \hline A & 11 & \wedge & 1 & 1 \\ & \barwedge & & \barwedge & \barwedge \\ B & 11 & \wedge & 1 & 1 \\ \hline & 1111 & \wedge & 11 & 0 \end{array} \qquad \begin{array}{c|rcr|c} & F_0 & & F_1 & \text{Result} \\ \hline A & 1111 & \wedge & 11 & 0 \\ & \barwedge & & \barwedge & \barwedge \\ B & 11 & \wedge & 1111 & 0 \\ \hline & 111111 & \wedge & 111111 & \mathbf{1} \end{array} \tag{2}$$

$$\begin{array}{c|rcr|c} & F_0 & & F_1 & \text{Result} \\ \hline A & 11 & \wedge & 11 & 0 \\ & \barwedge & & \barwedge & \barwedge \\ B & 1 & \wedge & 1 & 1 \\ \hline & 111 & \wedge & 111 & 1 \end{array} \qquad \begin{array}{c|rcr|c} & F_0 & & F_1 & \text{Result} \\ \hline A & 111 & \wedge & 111 & 1 \\ & \barwedge & & \barwedge & \barwedge \\ B & 111 & \wedge & 111 & 1 \\ \hline & 111111 & \wedge & 111111 & \mathbf{0} \end{array} \tag{3}$$

Since anything *NAND*ed with 0 results in 1, FRIBs uses 0 to maintain order. This is why the fragmentation is currently done with the *AND* function, so that if a server has a 0, it knows the bit value is *low*. Equations 4 and 5 give the same examples as in Equations 2 and 3, but now maintains order. Equation 4 now gives the right fragment value of 11011011 instead of 111111, which represents $11 \barwedge (11 \barwedge 11)$. But the left fragment value of 110110011 has an extra 0, giving $(11 \barwedge 11) \barwedge 11$ as the order is different to the right side. The second example in Equation 5 is more straightforward, as both sides are the same.

$$\begin{array}{c|rcr|c} & F_0 & & F_1 & \text{Result} \\ \hline A & 11 & \wedge & 1 & 1 \\ & \barwedge & & \barwedge & \barwedge \\ B & 11 & \wedge & 1 & 1 \\ \hline & 11011 & \wedge & 11 & 0 \end{array} \qquad \begin{array}{c|rcr|c} & F_0 & & F_1 & \text{Result} \\ \hline A & 11011 & \wedge & 11 & 0 \\ & \barwedge & & \barwedge & \barwedge \\ B & 11 & \wedge & 11011 & 0 \\ \hline & 110110011 & \wedge & 11011011 & \mathbf{1} \end{array} \tag{4}$$

$$\begin{array}{c|rcr|c} & F_0 & & F_1 & \text{Result} \\ \hline A & 11 & \wedge & 11 & 0 \\ & \barwedge & & \barwedge & \barwedge \\ B & 1 & \wedge & 1 & 1 \\ \hline & 1101 & \wedge & 1101 & 1 \end{array} \qquad \begin{array}{c|rcr|c} & F_0 & & F_1 & \text{Result} \\ \hline A & 1101 & \wedge & 1101 & 1 \\ & \barwedge & & \barwedge & \barwedge \\ B & 1101 & \wedge & 1101 & 1 \\ \hline & 1101001101 & \wedge & 1101001101 & \mathbf{0} \end{array} \tag{5}$$

C. Reduction

Reducing the size of a fragment requires information about all fragments. A seperate server, known as the reduction server, is used where all N servers send their fragments to during the reduction step. Once it has received each fragment, it uses a lookup table to retrieve the reduced fragments for each server. However if each fragment was sent to and returned from the reduction server in the current format, then some of the data can be decoded. The reduction servers have no knowledge of the program being run over the data, meaning any bits they can decode may still be worthless.

Since the reduction server is performing a simple lookup, we can obfuscate each fragment state to a unique value. For example, each server can hash the fragment with a server unique salt value, or use a random mapping. Now the reduction server cannot know the state of the fragments it has received, but protecting the reduced fragments is slightly more difficult. Instead of using a hash algorithm, we use public-key encryption on each reduced fragment, such that only the single server can decrypt the fragment. The lookup table is built offline and sent to the reduction server. Therefore the reduction server only receives a protected lookup table, and all the reduced fragments are already encrypted. Another security benefit given by this is that each public key for the individual servers and their reduced fragments, can remain private. To further improve privacy, multiple reduction servers can be used were each one is used in a pseudo-random order.

With a maximum fragment size of 32-bits, it produces >30000 entries per server. Using just two servers creates a very large multi-key lookup table. Reducing the fragment size to two sets of 16-bits for the lookup table, gives < 200 entries per server. This makes implementation more practical as the lookup table is now of a feasible size. With a maximum of <200 entries per server, each key only requires 8-bits, which can be increased by a few bits for better hashing. Splitting a fragment into two 16-bit values must be done at the last operation. For example, $110110011011000110110011011$ will become 110110011011 and 110110011011, allowing the same lookup table to be used to get two obfuscated values. Another lookup table can then get the encrypted values for each server using the two obfuscated values.

By using the same lookup table many times, we can increase the number of operations before the fragments need to be reduced, if the fragments have the available space to grow. Meaning that 110110011011 and 110110011011 can become four 16-bit values 110110011011, 110110011011, 110110011011 and 110110011011. Given that more servers lead to larger lookup tables, the size of the keys may need to reduced even further. Finding the balance between number of reductions and size of the fragments is important for performance, as described in Section VI-D.

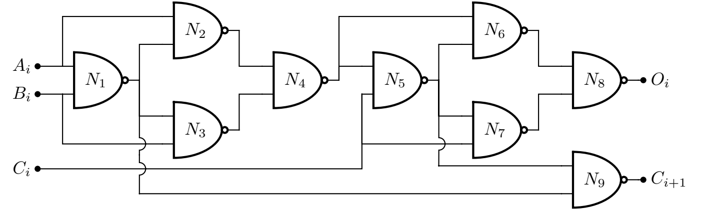

Fig. 2. NAND Gate Full Adder

VI. Implementation and Proven Practicality

A. Addition

The addition of two 32-bit integers can be achieved with thirty one full-adders and a single half-adder. A full-adder comprised of *NAND* gates can be seen in Figure 2. In order to get the best performance for our proposed scheme, we must reduce the number of network requests required by combining many reductions requests into a single request. First

we compute all values for N_4, which for worst-case where A_i and B_i are both 1, gives 10111001101. The fragment therefore can grow up to 10-bits during this step. We can then combine all 32 fragments for N_4 into a single network payload and send them to be reduced to single bits.

How the carry bits are reduced can vary depending on implementation, however we will allow the fragments to grow as large as needed for this step. If a limitation is applied, more reduction requests will be needed. Because the first bit does not have an input carry value, N_9 input is N_1 and N_1 (equates to $!N_1$). The other carry bits involve gates N_1, N_4, N_5 and N_9, where the result from N_9 is connected to the next bits N_5 gate. Given that N_4 will be a single bit, and that the worse case value for N_1 is 11, each carry step will at most add 5-bits to the fragment (11010). We only need a single 0 between each operation because we know the order is continuous. For example if the carry output for the second bit is 1101011011, we know the order of operations is $(110(10(110(11))))$.

This results in a worse-case fragment size of 155-bits ($16 \times$ 10-bit values). We then send these carry-bit fragments to be reduced, meaning we now have single bit values for all N_4 and N_9 gates. Allowing us to compute all N_8 gates with a maximum fragment size of 10-bits again. This only totals three reduction requests.

B. Multiplication

Binary multiplication can be thought of as a series of *AND* operations, all added together. Equation 6 shows an example of multiplying 5 and 11 on an 8-bit machine. For each bit in 11, we *AND* it with each bit in 5, giving 8 values. Adding each value together, gives 55.

$$\begin{array}{r} 00000101 \\ \times\ 00001011 \\ \hline 00000101 \\ 0000101 \\ 000000 \\ 00101 \\ 0000 \\ 000 \\ 00 \\ +\ 0 \\ \hline 00110111 \end{array} \tag{6}$$

To make the additions more efficient, we add together the biggest and next biggest values together, then the next pairing, down to the smallest and second smallest. This is shown in Equation 7.

$$\begin{array}{r} 00000101 \\ +\ 0000101 \\ \hline 00001111 \end{array} \quad \begin{array}{r} 0000 \\ +\ 000 \\ \hline 0000 \end{array} \quad \begin{array}{r} 000000 \\ +\ 00101 \\ \hline 001010 \end{array} \quad \begin{array}{r} 00 \\ +\ 0 \\ \hline 00 \end{array} \tag{7}$$

We repeat this step in Equation 8.

$$\begin{array}{r} 00001111 \\ +\ 001010 \\ \hline 00110111 \end{array} \quad \begin{array}{r} 0000 \\ +\ 00 \\ \hline 0000 \end{array} \tag{8}$$

And the final addition gives us the result in Equation 9.

$$\begin{array}{r} 00110111 \\ +\ 0000 \\ \hline 00110111 \end{array} \qquad \therefore = 00110111 \tag{9}$$

This gives a total of 7 addition operations for this example. But by adding similar sized numbers together in parallel, we decrease the number of reduction steps required. For Equation 7, each addition can combine the reduction requests into one, meaning the performance is close to that of a single addition. Therefore the performance of this example will be slightly above 3 additions. For 32-bit values, there are a total of 31 additions, but it performs like 5 additions.

C. Conditional

Supporting an operation to compare two values can dramatically affect the security of a secure processing scheme. For example if a group of cipher values only encrypted the set $\{0, 1\}$, then the ability to calculate if two cipher values are equal will result in two subgroups of cipher values. Where one subgroup must contain either encrypt a 0 or 1, and the other subgroup must encrypt the opposite. However because our proposed scheme has the bits fragmented across many servers, all the servers must compute over the same instruction set. This prevents a compromised sever trying to compare all the fragments it has, as the other fragment servers would need to be doing the same malicious action. Therefore our scheme has the ability to support conditional operations, which can be implemented to return the result in either a secure or non-secure manner.

1) Secure Results: Returning results securely, means the result is a fragmented bit, where < 1 fragment server has knowledge of the result. This can make some programs difficult to implement as the result of the comparison is not known. Two examples are given in Algorithms 1 and 2, for an equal and greater than or equal *if* statement. For both examples, we have to increment c without knowing the result of the comparison.

Algorithm 1 If Equals Example

1: **if** $a = b$ **then**
2: $\quad c \leftarrow c + 1$
3:
4: **function** IFEQUAL(a, b)
5: $\quad m \leftarrow a - b$
6: $\quad inout \leftarrow 0$
7: $\quad carry \leftarrow 0$
8: $\quad$ **for** $i \leftarrow 0$ to 32 **do**
9: $\quad\quad tmp \leftarrow m[i] + inout + carry$
10: $\quad\quad inout \leftarrow tmp \ \&\ 1$
11: $\quad\quad carry \leftarrow tmp >> 1$
12: $\quad$ **return** $!(inout \mid carry)$
13: $c \leftarrow c + (1 \times ifEqual(a, b))$

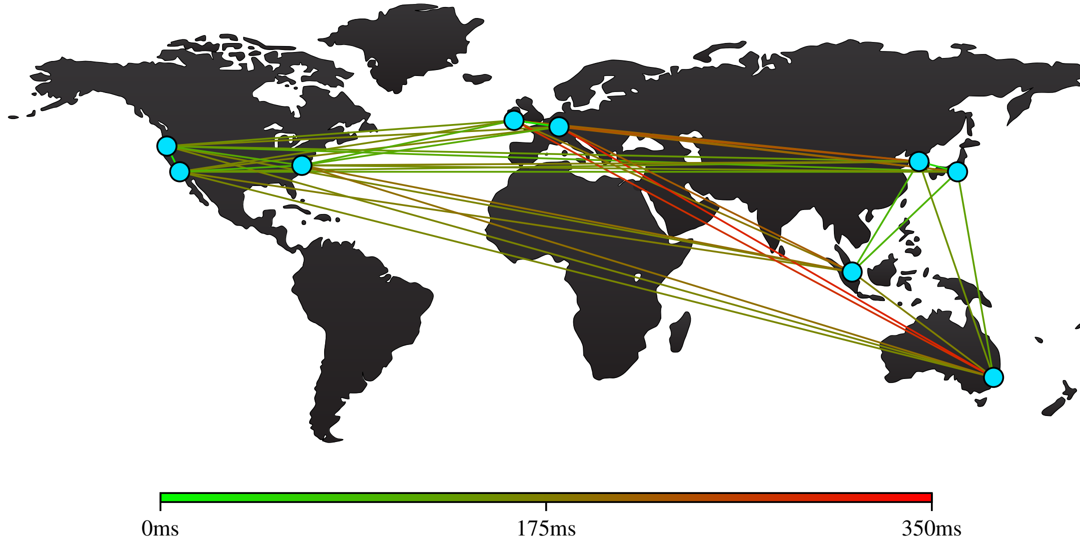

Fig. 3. Network latency experienced across 9 AWS datacenter locations.

Algorithm 2 If Greater Than or Equal Example

1: **if** $a >= b$ **then**
2: $\quad c \leftarrow c + 1$
3:
4: **function** IFGREATEREQUAL(a, b)
5: $\quad sign_neq \leftarrow a[31] \hat{} b[31]$
6: $\quad c \leftarrow a - b$
7: $\quad$ **return** $(!sign_neq \,\&\, !c[31]) \mid (sign_neq \,\&\, !a[31])$
8: $c \leftarrow c + (1 \times ifGreaterEqual(a, b))$

2) Non-Secure Results: Instead of returning a fragmented bit, this approach returns the whole bit by using a different lookup table than for a standard operation. This allows each server to know the result of the conditional statement, making programs easier to design and in some cases compute faster.

D. Performance

1) Network Latency: Because FRIBs relies heavily on reduction requests, the locations of the fragment/reduction servers affects performance. Figure 3 shows the average latency (round trip time) experienced during testing on Amazon AWS free-tier virtual machines in 9 locations: Virginia, California, Oregon, Ireland, Frankfurt, Singapore, Seoul, Tokyo and Sydney. Locations near each other had good connectivity, for example Singapore, Seoul and Tokyo, or Virginia, California and Oregon. In this test, Sydney did not perform well, with the best results over 100ms. This is something that needs to be considered when choosing server locations, as countries like Australia and New Zealand do not have the same level of connectivity as other Asia-Pacific countries [35].

2) Addition: The cloud service providers used for this experiment were Amazon Web Services, Microsoft Azure and Google Cloud Platform. All instances were running with the cheapest tier option, and based in the United States. The server configuration was a single reduction sever and two fragment servers. The reduction server was in California with Amazon, a fragment server was also in California but with Microsoft, and the final fragment server was in Iowa hosted by Google. We used a proof-of-concept addition algorithm with a 27-bit maximum fragment size which required 9 reductions, and averaged 100 addition operations for 32-bit unsigned integers. The latency at the time of testing was 3.106ms for Azure-Amazon, and 37.414ms for Google-Amazon.

Our results produced an average of 346ms for each addition operation. This is directionally proportional to the largest latency time, where $37.414 \times 9 \approx 346 -$ (some small computation times). Therefore if all the fragment servers could be within 10ms round trip from the reduction server, then addition times could be 99.274ms. The latency figures of Azure to Amazon could result in 37.228ms. Allowing for a larger fragment size would also increase the performance. For example, if only 5 reductions are required for an addition, then we can nearly half the completion time. These performance numbers are much faster than FHE schemes described in Section II-A.

3) Threading Issues: Each fragment server executes the same instructions before a reduction is needed, and all the fragments need to be received before the reduction server can return the results. Ideally all the servers are running the users thread at the same time. In reality, this cannot be guaranteed. We tested starting 50 user threads on 2 fragment servers in random order with a 10ms delay. Each user task computes 100 additions operations. Figure 4 shows 10% of the threads for the first few addition operations. Each line represents the start of an addition operation across both servers, where the color represents a unique user task. The labeled thread starts early on S_1, but takes longer to start on S_0. However the start of the next 5 additions happen at nearly the same time on both

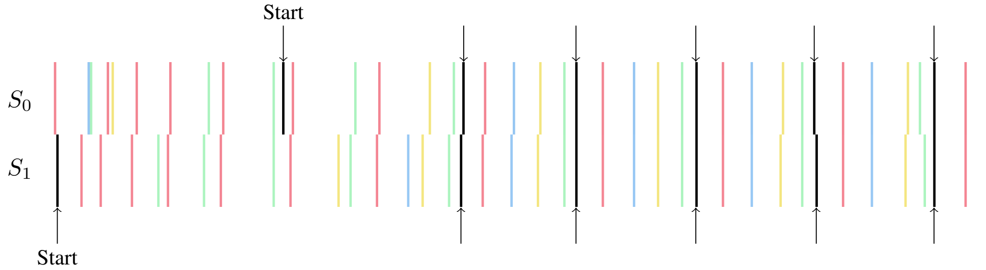

Fig. 4. Thread scheduling across 2 fragment servers for 50 randomly started user jobs.

servers. This happens as each thread must wait for the other to reach the reduction stage. Therefore, the threads are able to line up on both servers, where the thread counts on each are similar.

VII. Security Evaluation

Given our threat model in Section III-C, there are two main attack vectors we need to evaluate against: breaking the data with one set of fragments, and getting all the fragments from each system.

A. One set of Fragments

Successful attacks on traditional cryptography can often decrypt all the data [36][37], but with our scheme each bit fragment must be tried for each possibly. For example, we have two fragment servers and the bit fragments (b_{0-7}) for a byte known to be an ASCII value in the range of $32 - 122$. Because we only know $\approx 1/4$ of the bits, there are a number of possibilities for the value. If we know b_1 and b_3 to be zero, then we have 24 possibilities, of which 14 are letters. If we increase the number of fragment servers to four, then we might only know one bit. Setting only b_3 or b_1 to zero gives 48 or 46 possibilities respectively. Once we have hundreds of ASCII fragments, forming sentences and paragraphs, the number of possibilities are massive ($14^{100} \approx 4.1 \times 10^{114}$). Therefore privacy is preserved by the large amount of computation time to generate and test all possibilities. Note, if an attacker manages to break into more systems, then the number of possibilities will decrease as more bits are known.

B. Breaking into all Systems

Any service implementing our scheme is ultimately responsible for the security of their own environment. Software patching, access polices and firewall management, to name a few examples. However some exploits are out of a cloud services control, for example a zero-day vulnerability in their operating system. To reduce this risk, a mixture of Linux and Microsoft servers can be used, such that any one vulnerability cannot exploit every server. When a user/business is choosing the service providers, they should also seek information regarding security measures in place. A common approach is looking at a list of standards the service is compliant with.

International standards are now emerging for cloud service providers, with ISO/IEC 27018:2014 (with ISO/IEC 27002 as one of its normative references) being the first International code of practice that focuses on protection of Personally Identifiable Information (PII) in the cloud. This increases the security of their service, while providing more trust to their users. The cryptography recommendations/requirements described in ISO/IEC 27018:2014 are the objectives specified in clause 10 of ISO/IEC 27002. Examples are provided for use of cryptography in the cloud, but at a minimum a cloud should implement controls for confidentiality (data encrypted when stored or transmitted), and authentication. However there is no mention of true secure processing, like homomorphic encryption. To try and protect data being processed, access controls are recommended. This makes it more difficult for rogue employees or outside attackers to gain access to data in-flight. Therefore by conforming with ISO/IEC 27018:2014 will reduce the chance of a breach, and applying our scheme will enhance the security already provided.

End users also have control over their security, as the more fragment and reduction servers used, the smaller the risk of their data being compromised. Ten servers will give more security over five, but the running costs increase. Therefore evaluating against this attack vector is implementation dependant. FRIBs can protect data from rogue employees and malicious users who break into a few systems, but the cloud service providers need to try protect the fragments they store as well.

VIII. Applying Extra Security Layers

A. Hardware Implementation

For users/businesses requiring even more security, and are prepared to lose the flexibility of the cloud, hardware implementations are viable. A problem with secure processors like AEGIS [21], is they are vulnerable to side-channel-attacks as described in Section II-B. With non-secure values, if the hardware server is compromised, the data is compromised. However by having the bits themselves distributed, they only get $1/(2N)$ of the data required to reveal the value. Building a hardware server for our scheme would make the attacks physical, and therefore more difficult, especially when they are spread around the world.

B. Public-Key Encryption

One way to further protect the fragmented data is to have it encrypted. Note that when the data is in storage, the cloud service provider should encrypt it as well. But the user also has the ability to encrypt the data inside the fragments. For example, a business needs to share data with its customers. Each customer is sent fragments from each server in order to decode the data. There are two methods for securely sending the data to the customers. Each fragment server can encrypt the actual fragments before sending them to the user, or the data itself can be encrypted inside the fragments. The first method requires each server to have the customers public key, where the second method has the public key fragmented across each server.

Having the ability to store public keys within fragments gives them more protection. But it also means we can store private/decryption keys in the cloud. For example, after some data has been processed, it can encrypted within the fragments. Then when the data is needed again, we can decrypt it. This follows the same form of protection as described in Section II-B for secure processors.

C. Homomorphic Encryption

Like public-key encryption, homomorphic encryption could be applied within the fragments. This would allow encrypted data to be processed, while being distributed. The performance overhead of this would be large, if keys/dimensions are big. However because the data is already protected, we could make the keys/dimensions smaller.

IX. Use Cases and Comparisons

A. Small to Medium Sized Businesses

Many companies, in particular smaller or new businesses can benefit from the cloud, for example cost savings [38]. A problem emerges when personal customer data, or in house private data is stored in the cloud. Customers have an agreement with the company to store this information, but not directly with the cloud service provider. Even though the data would be visible to malicious employees of the cloud service provider. These problems have stopped some business from adopting the cloud, as they cannot guarantee the data is protected.

Unfortunately having data 100% protected while in the cloud is not currently possible. Even when meeting every standard, having the correct polices in place, and using the strongest encryption, someone, somewhere with the intent, resources and time will break it. The best we can do is get close to 100%. Like homomorphic encryption, FRIBs has the flexibility to meet companies required security requirements. However the stronger FRIBs becomes, by increasing the number of fragment and reduction servers, it still maintains similar performance. Where with homomorphic encryption, as the key/dimension size increases, so does the computation times.

B. Internet of Things: Low Energy and Low Powered Devices

The Internet is spreading to all of our devices, from kitchen appliances, to autonomous cars and drones. Many of these devices have very little computation power, and can often be powered by a battery. The challenge is protecting the data they collect and transmit. Traditional encryption techniques are too inefficient, meaning security is often overlooked in favour of functionality

Wireless sensor networks [39] are becoming widely deployed [40] in commercial, military and personal environments. Schemes have been proposed to encrypt data [40][41] efficiently on wireless sensor nodes, however the data must be decrypted before any processing can be applied. Using FHE, an Intel Core i7 3770K at 3.5GHz takes 1.08 seconds and 10.6 seconds for the encryption of a small and medium sized dimension [14]. Where a sample sensor node using an Intel StrongARM SA-1110 microprocessor, only has a frequency of 59MHz to 206MHz [42] (relatively fast compared to other sensor node processors). Combined with other architecture differences, like data-path and caches sizes, the sensor node cannot encrypt using a FHE scheme in feasible time. Where FRIBs requires very little effort to fragment bits, and can utilise existing encryption schemes [40][41] built for sensor networks to send the fragments. This also reduces the amount of data the nodes have to send, as FHE can produce large cipher-values.

X. Conclusion

This paper has described FRIBs, a novel scheme to compute arbitrary operations in the cloud while preserving user privacy. Using a different methodology to state-of-the-art solutions, individual bits are fragmented and spread across different cloud service providers, rendering the values incomprehensible. FRIBs follows the idea of "hiding in the masses", as proposed in [31], where each bit fragment has a large number of possibilities.

One future possibility for FRIBs is to use a different technique for fragmenting the bits. This is to prevent $1/(2N)$ of the data being visible to each fragment server. We would also like to conduct a more in-depth security analysis of FRIBs, for example observing patterns of hashes received by the reduction server. Others include looking into secure instruction sets, improving the multiplication approach, supporting dynamic memory lookups and more conditional statements. Combined with increasing performance, a distributed secure virtual processor becomes plausible.

FRIBs computes operations in a fraction of the time as the holy grail of cryptography (i.e. fully homomorphic encryption), reducing processing time from hours to seconds. This is achieved by shifting the overhead from computation, to network latency between each fragment server and the reduction server/s. By allowing for varying performance and security, users can now take control of their data in the cloud.

Acknowledgements

This research is supported by STRATUS (Security Technologies Returning Accountability, Trust and User-Centric Services in the Cloud) (https://stratus.org.nz), a science investment project funded by the New Zealand Ministry of Business, Innovation and Employment (MBIE). The authors would like to acknowledge Dr Sivadon Chaisiri, and fellow members of the CROW lab (https://crow.org.nz) for their assistance in the manuscript review.

References

[1] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On data banks and privacy homomorphisms," *Foundations of secure computation*, vol. 4, no. 11, pp. 169–180, 1978.

[2] R. K. Ko, "Cloud computing in plain english," *ACM Crossroads*, vol. 16, no. 3, pp. 5–6, ACM, 2010.

[3] "Dropbox," Online [Accessed 15/05/16] https://www.dropbox.com.

[4] A. Chen, "Google Engineer Stalked Teens, Spied on Chats," Online [Accessed 26/08/14] http://gawker.com/5637234/gcreep-google-engineer-stalked-teens-spied-on-chats, Gawker, September 2010.

[5] "How secure is Dropbox?" Online [Accessed 15/04/16] https://web.archive.org/web/20140121013018/https://www.dropbox.com/help/27/en, January 2014.

[6] "Mega Limited," Online [Accessed 15/04/16] https://mega.co.nz.

[7] M. A. Will and R. K. Ko, "A guide to homomorphic encryption," in *The Cloud Security Ecosystem: Technical, Legal, Business and Management Issues*. Elsevier, 2015, pp. 101–127.

[8] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, pp. 120–126, 1978.

[9] M. Hirt and K. Sako, "Efficient receipt-free voting based on homomorphic encryption," in *Advances in Cryptology—EUROCRYPT 2000*. Springer, 2000, pp. 539–556.

[10] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: protecting confidentiality with encrypted query processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 85–100.

[11] D. Boneh, C. Gentry, S. Halevi, F. Wang, and D. J. Wu, "Private database queries using somewhat homomorphic encryption," in *Applied Cryptography and Network Security*. Springer, 2013, pp. 102–118.

[12] M. A. Will, B. Nicholson, M. Tiehuis, and R. K. Ko, "Secure Voting in the Cloud using Homomorphic Encryption and Mobile Agents," in *International Conference on Cloud Computing Research and Innovation*, vol. 1. IEEE, 2015, pp. 173–184.

[13] C. Gentry *et al.*, "Fully homomorphic encryption using ideal lattices." in *STOC*, vol. 9, 2009, pp. 169–178.

[14] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Exploring the feasibility of fully homomorphic encryption," *Computers, IEEE Transactions on*, vol. 64, no. 3, pp. 698–706, 2015.

[15] C. Gentry and S. Halevi, "Implementing gentry's fully-homomorphic encryption scheme," in *Advances in Cryptology–EUROCRYPT 2011*. Springer, 2011, pp. 129–148.

[16] X. Cao, C. Moore, M. O'Neill, N. Hanley, and E. O'Sullivan, "High-speed fully homomorphic encryption over the integers," in *Financial Cryptography and Data Security*. Springer, 2014, pp. 169–180.

[17] M. Hirvensalo, *Quantum computing*. Springer, 2001.

[18] S. Barz, E. Kashefi, A. Broadbent, J. F. Fitzsimons, A. Zeilinger, and P. Walther, "Demonstration of blind quantum computing," *Science*, vol. 335, no. 6066, pp. 303–308, 2012.

[19] W. Diffie and M. E. Hellman, "New directions in cryptography," *Information Theory, IEEE Transactions on*, vol. 22, no. 6, pp. 644–654, 1976.

[20] D. Hankerson, A. J. Menezes, and S. Vanstone, *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.

[21] G. E. Suh, C. W. O'Donnell, and S. Devadas, "AEGIS: A single-chip secure processor," *Information Security Technical Report*, vol. 10, no. 2, pp. 63–73, 2005.

[22] B. Yang, K. Wu, and R. Karri, "Scan based side channel attack on dedicated hardware implementations of data encryption standard," in *International Test Conference, 2004. Proceedings. ITC 2004.* IEEE, 2004, pp. 339–344.

[23] B. Köpf and D. Basin, "An information-theoretic model for adaptive side-channel attacks," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 286–296.

[24] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology—CRYPTO'99*. Springer, 1999, pp. 388–397.

[25] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: Concrete results," in *Cryptographic Hardware and Embedded Systems—CHES 2001*. Springer, 2001, pp. 251–261.

[26] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," *ACM SIGCOMM computer communication review*, vol. 39, no. 1, pp. 68–73, 2008.

[27] G. Xu, J. Pang, and X. Fu, "A load balancing model based on cloud partitioning for the public cloud," *Tsinghua Science and Technology*, vol. 18, no. 1, pp. 34–39, 2013.

[28] V. Ganapathy, D. Thomas, T. Feder, H. Garcia-Molina, and R. Motwani, "Distributing data for secure database services," in *Proceedings of the 4th International Workshop on Privacy and Anonymity in the Information Society*. ACM, 2011, p. 8.

[29] R. Cramer, I. Damgård, and J. B. Nielsen, *Multiparty computation from threshold homomorphic encryption*. Springer, 2001.

[30] B. Adida, "Helios: Web-based open-audit voting." in *USENIX Security Symposium*, vol. 17, 2008, pp. 335–348.

[31] M. A. Will, R. K. Ko, and I. H. Witten, "Bin Encoding: A User-Centric Secure Full-Text Searching Scheme for the Cloud," in *Trustcom/BigDataSE/ISPA, 2015 IEEE*, vol. 1. IEEE, 2015, pp. 563–570.

[32] "Amazon Web Services, Inc." Online [Accessed 07/04/16] https://aws.amazon.com.

[33] "Microsoft Azure," Online [Accessed 07/04/16] https://azure.microsoft.com.

[34] B. G. Streetman and S. Banerjee, *Solid state electronic devices*. Prentice Hall New Jersey, 2000, vol. 5.

[35] R. Pfeffer, "International Submarine Cables and Hi-Speed Broadband in Australia and New Zealand," Southern Cross Cables Limited, Tech. Rep., 2010.

[36] D. Boneh, "Twenty years of attacks on the RSA cryptosystem," *Notices of the AMS*, vol. 46, no. 2, pp. 203–213, 1999.

[37] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 305–316.

[38] S. Marston, Z. Li, S. Bandyopadhyay, J. Zhang, and A. Ghalsasi, "Cloud computing—the business perspective," *Decision support systems*, vol. 51, no. 1, pp. 176–189, 2011.

[39] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "A survey on sensor networks," *Communications magazine, IEEE*, vol. 40, no. 8, pp. 102–114, 2002.

[40] R. Watro, D. Kong, S.-f. Cuti, C. Gardiner, C. Lynn, and P. Kruus, "Tinypk: securing sensor networks with public key technology," in *Proceedings of the 2nd ACM workshop on Security of ad hoc and sensor networks*. ACM, 2004, pp. 59–64.

[41] A. Perrig, R. Szewczyk, J. D. Tygar, V. Wen, and D. E. Culler, "SPINS: Security protocols for sensor networks," *Wireless networks*, vol. 8, no. 5, pp. 521–534, 2002.

[42] E. Shih, S.-H. Cho, N. Ickes, R. Min, A. Sinha, A. Wang, and A. Chandrakasan, "Physical layer driven protocol and algorithm design for energy-efficient wireless sensor networks," in *Proceedings of the 7th annual international conference on Mobile computing and networking*. ACM, 2001, pp. 272–287.