Privacy-Preserving Encoding for Cloud Computing

A thesis
submitted in fulfilment
of the requirements for the degree
of
Doctor of Philosophy

at
The University of Waikato

by
Mark A. Will

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

Department of Computer Science
Hamilton, New Zealand

Abstract

Information in the cloud is under constant attack from cyber criminals as profitability increases; user privacy is also at risk with data being mined for monetary value – the new gold. A single leak could have devastating consequences for a person or organisation, yet users have limited control over their privacy. It is becoming clear that the current model for public cloud computing is flawed, where cloud vendors and their employees can no longer be trusted to protect user data. Privacy-preserving computation in the cloud keeps data private at all times but still remains functional, thus returning control of data back to users. The cloud could then perform operations using data that it cannot comprehend. The end-user would then be able to retrieve the results from the cloud and unlock the real answers.

Homomorphic encryption is a solution for privacy-preserving processing, allowing computation over cipher text. At the time of writing, a fully homomorphic system allows arbitrary operations but requires minutes to compute an operation, whereas partially homomorphic encryption can only support a single operation, meaning it cannot be a generic solution to privacy-preserving computing. Another solution is multi-party computation, which uses a distributed approach built upon homomorphic encryption but currently suffers other limitations like reusability and lacks the ability to be truly dynamic.

The primary objective of this research is to design a solution for the cloud that offers privacy-preserving data computation but provides performance and flexibility. A novel approach for multi-party computation is developed, where the combination of encoding and distribution helps provide the balance between security, performance and utility. Privacy is maintained by each distributed entity only receiving a small portion of the actual data through encoding, where attempting to brute-force the data results in a vast number of

possibilities, similar to encryption. Functions are defined with universal or custom logic and are computed quickly, as the performance overhead is no longer computational but network latency. A cloud voting application was used for analysis between existing solutions and the novel approach taken by this research, which is able to add thousands of votes per minute, giving practical privacy-preserving processing in the cloud.

Acknowledgements

This has been the biggest challenge of my life thus far; at times the end seemed impossible, the fear of failure ever looming, with those words of "why did I do this" getting stronger. However, the love and support of friends, family and colleagues has pushed me through to this point. I feel as though I have grown as a researcher, but also as a person and I have everyone to thank for that.

Throughout my studies, I have been lucky enough to receive the following scholarships: University of Waikato Doctoral Scholarship, Top Achiever Doctoral Scholarship, a STRATUS Research Scholarship and the Prime Minister's Scholarship for Asia, which all made a big difference and I am very grateful. I have also had support from the STRATUS project, and was able to work as a Doctoral Assistant for four years, which I cannot thank the department enough for. On that note, the computer science department has been amazing throughout my eight years studying at Waikato, and I can safely say I would not be where I am today without that unique environment.

There have been many positives during my studies, including traveling around the world, experiencing other cultures, and many other opportunities. The biggest of which, was having the chance to work at INTERPOL in Singapore for three months. It was a true privilege, and there are many people to thank for making that possible. Toshinobu Yasuhira, Margaret Samuel, Silvino Schlickmann and the rest of the team, thank you for making my time there so memorable. However, there is one man who was responsible for all my amazing opportunities and experiences; thank you Ryan, I know you probably did not hear that enough, but I have always been grateful for your hard work and determination to push us to be the best we can.

Finally, I would like to single out a few people who have helped me along the way. Ian and Jim, my thesis would be useless without you guys, I owe

you both. To the lab, we had something special and I know I will never find that again; thank you Alan, Jeff, Sam, Craig, Baden, Boom, Brandon, Harris, Raja, Mohammad and Brad. It is actually quite emotional just seeing your names here, and I think that really signifies the relationship we all had. Alan and Jeff, we sure had some fun together, I really cannot thank the two of you enough, which is why you got mentioned twice. Finally, I would like to thank my wife, for getting me through this final stage, for putting up with me during what has been stressful times, and for making everything better again, mahal kita.

I am sorry if I have missed anyone, you have all helped me get through this, salamat!

List of Publications

2017

- **Secure FPGA as a Service – Towards Secure Data Processing by Physicalizing the Cloud**, **Mark A. Will** and Ryan K. L. Ko, The 16th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Sydney, NSW, 2017 **(CORE Rank A)**

- **Anonymous Data Sharing Between Organisations with Elliptic Curve Cryptography**, **Mark A. Will**, Ryan K. L. Ko and Silvino J. Schlickmann, The 3rd IEEE International Workshop on Trust and Security in Wireless Sensor Networks (TrustWSN) at TrustCom, Sydney, NSW, 2017

- **Visualizing the New Zealand Cyber Security Challenge for Attack Behaviours**, Jeffery Garae, Ryan K. L. Ko, Janice Kho, Saidah Suwadi, **Mark A. Will** and Mark Apperley, The 3rd IEEE International Workshop on Cloud Security and Forensics (WCSF) at TrustCom, Sydney, NSW, 2017

- **Returning Control of Data to Users with a Personal Information Crunch – A Position Paper**, **Mark A. Will**, Jeffery Garae, Yu Shyang Tan, Craig Scoon and Ryan K. L. Ko, International Conference on Cloud Computing Research and Innovation (ICCCRI), Singapore, 2017, **Best Paper Award**

- **Chapter 5 - Distributing Encoded Data for Private Processing in the Cloud**, **Mark A. Will** and Ryan K. L. Ko, Data Security in Cloud Computing, IET, 2017

2016

- **Privacy Preserving Computation by Fragmenting Individual Bits and Distributing Gates**, **Mark A. Will**, Ryan K. L. Ko and Ian H. Witten, The 15th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Tianjin, 2016 **(CORE Rank A)**

- **Computing Mod with a Variable Lookup Table**, **Mark A. Will** and Ryan K. L. Ko, International Symposium on Security in Computing and Communication (SSCC 2016), Communications in Computer and Information Science, vol 625. Springer, Singapore

2015

- **Secure Voting in the Cloud using Homomorphic Encryption and Mobile Agents**, **Mark A. Will**, Brandon Nicholson, Marc Tiehuis and Ryan K. L. Ko, International Conference on Cloud Computing Research and Innovation (ICCCRI 2015), Singapore, 2015, **Best Paper Award**

- **Bin Encoding: A User-Centric Secure Full-Text Searching Scheme for the Cloud**, **Mark A. Will**, Ryan K. L. Ko and Ian H. Witten, The 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), Helsinki, 2015 **(CORE Rank A)**

- **Chapter 5 – A Guide to Homomorphic Encryption**, **Mark A. Will** and Ryan K. L. Ko, The Cloud Security Ecosystem, 1st Edition, Technical, Legal, Business and Management Issues, Syngress, an Imprint of Elsevier, 2015

2014

- **Progger: An Efficient, Tamper-Evident Kernel-Space Logger for Cloud Data Provenance Tracking**, Ryan K. L. Ko and **Mark A. Will**, The 7th IEEE International Conference on Cloud Computing (CLOUD), Anchorage, AK, 2014

Table of Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

AES	Advanced Encryption Standard
AWS	Amazon Web Services
CVP	Closest Vector Problem
ECC	Elliptic Curve Cryptography
FHEW	Fastest Homomorphic Encryption in the West
FPGA	Field-Programmable Gate Array
FRIBs	Fragmenting Individual Bits
GCD	Greatest Common Divisor
IoT	Internet of Things
LUT	Lookup Table
LWE	Learning With Errors
MTU	Maximum Transmission Unit
NAND	Negative-AND
PPE	Privacy-Preserving Encoding
PUF	Physical Unclonable Function
RSA	Rivest-Shamir-Adleman cryptosystem
RTT	Round Trip Time

SVP Shortest Vector Problem

TCP Transmission Control Protocol

XOR Exclusive-OR

1

Introduction

Placing responsibility for keeping data private entirely within the hands of a cloud vendor is not a sustainable solution to the general problem of data privacy [1][2]. This is supported by 88% of the 3,000 people surveyed being worried about who has access to their data, and 84% worrying about where this data is stored [3]. Furthermore, 91% would like a system which enables them to have control over their data [3]. Combined with the increasing profitability of data breaches for malicious entities, there is a need to protect data for its entire duration in the cloud. However, systems that protect data privacy today either have limited functionality [4][5][6] or are impractical [7][8][9][10][11].[1] The research presented in this thesis addresses this issue, by studying the problem from an implementation and practicality point-of-view.

1.1 Background

Encryption is a common approach to protecting data in the cloud. However, most cryptosystems preclude any computation over the data unless it is decrypted. In contrast, homomorphic encryption allows operations to be computed over cipher text [4]. Therefore, data can be protected while maintaining some functionality.

The term homomorphic originates from ancient Greek, where *homos* means same and *morphe* means shape. There are two types of homomorphic en-

[1] This thesis addresses privacy at the same level as encryption, where data should not be visible to the computing platforms; a layer below areas such as differential privacy [12][13].

cryption: *partial* and *full*. Partially homomorphic encryption can only support a single operation over cipher text, for example addition [6] or multiplication [4][5], where fully homomorphic encryption, also known as the *"holy grail"* of encryption, can support multipliable operations [9], allowing more advanced computation to be performed over encrypted data. However, currently no fully homomorphic encryption scheme is efficient enough to be used in the real world [14], and partially homomorphic encryption can be too limited for many applications.

The other state-of-the-art concept for privately processing data in the cloud is secure multi-party computation [15][16][17]. Instead of the data being protected on a single entity, the data is distributed across multiple parties, allowing them to jointly compute a function while keeping their inputs private from the other parties. Where fully homomorphic encryption is a computably intensive operation, multi-party computation suffers from large network transfers and implementation issues including reusability of garbled circuits, which is also discussed in Chapter 3. Many secure multi-party computation schemes make use of homomorphic encryption [18][19][20]; thus, in cases, the performance is also limited by that of the homomorphic scheme it is built upon.

Protecting data in the cloud faces other risks aside from practicality. Some governments today are looking at the possible ban on encryption or forcing the inclusion of backdoors. This is because encryption schemes are designed to protect the data they obfuscate, no matter who encrypted it. These political challenges could significantly weaken the security levels we have today. A different approach to encryption for protecting data should be considered to address this possible issue.

The research reported in this thesis removes the need for data to be encrypted (using a traditional scheme) for processing while still remaining protected. A custom secure processor [4][21] would also achieve this but is not widely deployable in the cloud, and again it still reveals the plain text inside the processor. One technique to keep the data private is with lossy encoding, where even when half the data is broken, the other half still needs to be cracked [22]. Another is by one entity not possessing the necessary data to view the plain-text [23].

1.2 Hypothesis

The operation limitations of partially homomorphic encryption, and computational intensity of fully homomorphic encryption, currently prevent homomorphic encryption from being usable for most cloud applications. The issues arise from the complexity of the encryption schemes and underlying mathematics, which we discuss in Chapter 2, leading to the primary research question for this thesis:

"Can fully homomorphic encryption be practical in the cloud?"

There were three steps taken to answer this question, and are given as the following questions, leading to the following hypotheses:

Hypothesis 1 *The underlying large number operations for homomorphic encryption can be optimised for better performance.*
Did not lead to major breakthroughs within the time of the PhD – see Appendix A.

Hypothesis 2 *A simpler fully homomorphic scheme can provide practical performance while remaining secure.*
This was unsuccessful due to security weaknesses of using simultaneous equations – see Appendix B.

Hypothesis 3 *Data need not be encrypted for practical privacy-preserving computation; instead, keyless obfuscation and distribution can protect privacy by a single entity not possessing enough information to decode the data, while supporting arbitrary computation.*
This hypothesis was formulated after 1 and 2 had been investigated and developed into the main hypothesis of this thesis.

1.3 Scope

1.3.1 Use Cases

Four use cases were considered in this thesis:

1. An entity would like to have a secure ballot, where votes must be private and only the final tally should be revealed in plain text. Therefore, information about how someone voted should not be obtainable.

2. A personal user would like to store their health/home data (heartbeat, sleep patterns, or room temperatures) collected from activity trackers and other Internet of Things (IoT) devices in the cloud. This data should remain private, but it needs to be processed to find alerts or information such as irregular heartbeats.

3. An organisation or company wants to store and process their confidential data in the cloud, where it needs to be kept private at all times, even from the cloud hosting company and its employees.

4. Multiple organisations would like to share data between them in the cloud but control the operations computed over them and be able to control access to their shared data (for example deleting data).

The primary use case was cloud voting because it allowed for comparison between state-of-the-art works. Use cases 2 and 3 are essentially the same as use case 1 in that operations need to be computed over private data. The difference is that use case 1 has many clients and requires high performance. The last use case targets controlling data in the cloud: who has access and what operations they can perform.

1.3.2 Privacy-Preserving Encoding

The schemes presented in this thesis do not claim to be encryption techniques; instead, they provide privacy through encoding. The term *Privacy-Preserving Encoding* is used to differentiate encoding from encryption, while still protecting data. The difference between encoding and encryption is minuscule [24].

Both transform data, but encryption keeps some parts of the process a secret. Without this secret value (the key), it should be computationally intensive to break. Therefore, if only the intended parties have the secret, then the data should remain hidden from everyone else. Note that hashing is usually for integrity, it takes an arbitrary input and returns a fixed-length "unique" output [24]; whereas obfuscation is very similar to encoding and for the purpose of this thesis, we will say they are the same.

Traditional cryptography follows the principle given below, by the Dutch cryptographer Auguste Kerckhoffs [25].

Principle *A cryptosystem must not depend on keeping the algorithm safe, only the secret key [25].*

However, encoding a value b into 3 values using Exclusive-OR (XOR) logic gates does not have a secret key. For example, with $b = 0$, an encoding could be $1 \oplus 0 \oplus 1$. If two of the values are known, b is still hidden. But this is not achieved with a secret key like traditional encryption.

One of the oldest encryption techniques known is the Caesar cipher, named after Julius Caesar who used it to protect messages of high importance, such as military instructions. This is a type of substitution cipher, which maps plain-text values to their cipher-text counterparts at a shifted index in the same alphabet. Decoding requires the shift number, which is the secret key. Mary, Queen of Scots, used a simple substitution cipher to try and plan the assassination of Queen Elizabeth I. This cipher mapped the alphabet and common words to another alphabet. For this scheme, the mapping table is classified as the secret. By today's standards, both these old techniques are weak, making it hard to classify them as encryption schemes. The nature of the algorithms are a shift index or a simple mapping, where most encryption schemes involve some hard mathematical problem, but they can still be categorised as encoding.

In legalisation and regulations, encryption is also often defined as something that uses a key. For example, in a letter from the New Zealand Ministry of Foreign Affairs and Trade in February 1997 states *"The export of code in any form is regulated in New Zealand in terms of the guidelines below: They contain encryption limited to (i) 40-bit key lengths for symmetric algorithms;*

(ii) 512 bits for asymmetric algorithms; (iii) 56-bit DES for dedicated financial algorithms" [26][27]. Therefore, for this thesis, the term *Privacy-Preserving Encoding* will be used to define a mapping-based cryptosystem, where the map is secret or defines a scheme that does not technically have a secret key but still provides data privacy.

1.3.3 Data Security and Privacy

Achieving perfect security for data storage and processing in the cloud does not guarantee the protection of data. It remains at risk because at some point the user will need to access the plain-text data. Even if the service requires authentication, a malicious user or a piece of malware can wait until access is granted. Mobile malware is a growing threat [28] and gives attackers the ability to steal data from the device. However data can also be stolen from services the device has access, for example a cloud service. This is a current open problem for applications like cloud voting with mobile agents [29], because currently security cannot be guaranteed on the mobile device. Similarly, the cloud is under threat from new attacks and vulnerabilities found each day (zero-day attacks) [30][31].

This thesis attempts to mitigate the risk of zero-days by spreading data across different cloud services. Meaning any scheme presented should be easy to implement on current cloud infrastructure, for example Amazon Web Services (AWS) [32] or Microsoft Azure [33], and not require any special hardware or equipment. Thus allowing it to be distributed across varying operating systems, applications and companies (to prevent a malicious employee from accessing data).

Data is considered to be private in the cloud when no single hosting entity can see the corresponding plain text. For example, if some data is hosted on cloud A, then if cloud A is compromised, the data is still obfuscated. However, if all hosts of some distributed data are compromised, the data could be lost. This is still considered private data for the purpose of this thesis. Note that this thesis investigates protecting data in the cloud, not necessary the operations being computed over it.

Similarly, this thesis assumes all parties perform the operations currently. No tampering, corrupting of the data or program being computed. We aim to

protect privacy from entities able to observe and copy data (eyes only). Some work was done to address this issue, but it is not the primary focus of this thesis. Our focus was instead on performance while keeping the data private.

Finally, for the purpose of this thesis data privacy is for computing nodes, where data outside of the computing node still requires the same protection mechanisms seen today. Even though Hypothesis 3 states that data will not be encrypted, this only targets the processing of the data. Data transfers across the network still need to be encrypted using current cryptography schemes, and data storage should still use encryption to provide an extra layer of security.

1.4 Contributions

The first contribution is purely as an implementation to see how existing works can create a practical privacy-preserving cloud application. This thesis has two main contributions, the schemes Bin Encoding [22] and Fragmenting Individual Bits (FRIBs) [23]. These use encoding techniques instead of encryption to provide data privacy in the cloud. They both protect data by *"hiding in the masses"* where there exist a large number of combinations and it is difficult to know when a correct solution is found.

1.4.1 Secure Survey Platform

In order to answer the research question, a definition and understanding of practicality for privacy-preserving cloud computation was required. This was accomplished by designing and implementing a secure survey platform using partially homomorphic encryption, presented in Section 3.3. Initially this was published as a voting scheme [29], before being converted to a survey system. In addressing some of the challenges faced, a fully secure cloud survey application is possible today, however with missing functionality and security concerns. The main contribution of this application was it shows the practicality of using privacy-preserving schemes in the real-world, from limitations with JavaScript and mobile devices on the client-side, to reduced functionality in the cloud which results in more processing by the survey creator.

1.4.2 A Private Encoding Search Scheme

Bin Encoding was initially designed as an attempt to help answer Hypothesis 3, where encoded data can provide some form of private functionality [22]. The private search functionality achieved is near that of plain text, as is performance, including client-side. Related work focuses on encryption and cryptographic trapdoor functions [34][35], where this thesis will show how a simple lossy encoding scheme can protect privacy for approximate string searching. However, Bin Encoding highlighted a few challenges with encoding data using a static mapping, one of which is the lack of randomness. Even so, Hypothesis 3 showed merit as even with knowledge of the mapping, it can still be difficult to recreate the plain-text values. A distributed model for Bin Encoding was also presented, where greater privacy was achieved by distributing the search across multiple cloud providers or services. The model for Bin Encoding was published [22], where Chapter 4 also gives some additional results.

1.4.3 FRIBs: An Arbitrary Privacy-Preserving Scheme

Bin Encoding showed the potential for using encoding for privacy-preserving processing. But a new scheme was needed to provide arbitrary computation in order to fully meet the requirements of the research question. Fragmenting Individual Bits (FRIBs) is presented in this thesis as a potential solution for a practical privacy-preserving processing scheme. It was designed to meet the hypotheses, but more importantly to try and answer the research question. The first model given in Chapter 5 was published (the reduction server model) [23], where the main models–the performance and enhanced privacy models–are the main contribution of this thesis. The design resembles multi-party computation but handles data sharing and garbled circuits using encoding. One advantage over other multi-party computation designs is that Fragmenting Individual Bits (FRIBs) can reuse its Lookup Table (LUT)s.

1.5 Organisation

The presentation of this thesis will start with current works. A review and discussion are given in Chapter 2, highlighting the problem of practicality. This

chapter identities that the idea of secure computation was proposed since early encryption schemes like the Rivest-Shamir-Adleman cryptosystem. The state-of-the-art work from Chapter 2 is implemented and compared in Chapter 3 for the use case of cloud voting. This helped create a definition for *"practical"*; a requirement when researching towards a practical solution. To summarise, practicality is a balance between performance, and flexibility or utility.

In researching Hypothesis 2, a discovery was made for appropriate privacy-preserving string searching using *"bins"*. Chapter 4 describes this research and gives analysis on the privacy achieved. The scheme uses a lossy random-mapping between an alphabet and a smaller alphabet. There are a large number of combinations for decoding, where knowing the correct answer is difficult. Even when successfully decoding half of the data, there are still a large number of combinations for the remaining half. Chapter 4 not only introduces Privacy-Preserving Encoding (PPE), but also the use of distribution to improve privacy as each server has less information.

The three Chapters 5, 6 and 7 present the main contribution of this thesis– the Fragmenting Individual Bits (FRIBs) scheme–addressing all four use cases and Hypothesis 3. The two core FRIBs models are described in Chapter 5: the performance model and enhanced privacy model. Both build off earlier work published on FRIBs [23], distributing single bits across multiple servers, while also managing to distribute the computation table (similar to a garbled circuit), such that no information is leaked during execution. Functions are defined like garbled circuits for multi-party computation, except that they can be reused as Chapter 6 demonstrates. The data can only be decoded by retrieving all the distributed bits, thus privacy and security is obtained by the multiple servers spread over different entities. Chapter 7 gives two proof-of-concept implementations: cloud voting and exact string searching; before evaluating their practicality. Privacy is then analysed in Chapter 7, proving that no information is leaked during processing when using the enhanced privacy model.

The hypotheses and use cases are revisited in Chapter 8, concluding that PPE, in particular FRIBs, can provide an alternative to computationally intensive encryption schemes. Chapter 8 also mentions future work, providing a few research directions for FRIBs; however, ultimately the target for FRIBs is being deployed in real-world implementations.

2

Literature Review

In this chapter, literature is reviewed related to privacy-preserving processing: in particular, works focused on cloud implementations. Chapter 1 briefly introduced homomorphic encryption and secure multiparty computation, which will be expanded further in Sections 2.1 and 2.2 respectively. Another popular method of privacy-preserving processing is with custom hardware processors, discussed in Section 2.3. However, it is worth noting that their effectiveness for cloud deployments is yet to be discovered [36].

2.1 Homomorphic Encryption

Homomorphic encryption is the current state-of-the-art solution to privacy-preserving processing in the cloud. As previously mentioned in Chapter 1 there are two types: partially and fully homomorphic encryption. Partially homomorphic encryption has faster computation but only supports a single operation, while fully homomorphic encryption is currently impractical but supports arbitrary computation [37]. Note that there is a third variate called somewhat homomorphic encryption, where only a limited number of operations can be applied before the scheme produces invalid results [38][39]. Many fully homomorphic encryption schemes actually have this limitation, but are able to re-encrypt (known as bootstrapping [39]) the cipher values, resetting the number of operations applied [8][9][39]. Given the use cases in Section 1.3.1, especially voting which requires millions of operations, somewhat homomorphic encryption will not be detailed in this thesis.

Table 2.1: Historic overview of homomorphic encryption

2013	Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based [9]
2010	Fully Homomorphic Encryption over the Integers [8]
2009	A Fully Homomorphic Encryption Scheme [39]
	Fully Homomorphic Encryption using Ideal Lattices [7]
2008	A New Approach for Algebraically Homomorphic Encryption [40]
2002	A Provably Secure Additive and Multiplicative Privacy Homomorphism [41]
1999	Public-Key Cryptosystems Based on Composite Degree Residuosity Classes [6]
1996	A New Privacy Homomorphism and Applications [42]
1988	On Privacy Homomorphisms [43]
1987	Processing Encrypted Data [44]
1985	A Database Encryption Scheme Which Allows the Computation of Statistics using Encrypted Data [45]
	A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms [5]
1982	Signature Protocols for RSA and Other Public-Key Cryptosystems [46]
	Probabilistic Encryption & How To Play Mental Poker Keeping Secret All Partial Information [47]
1978	On Data Banks and Privacy Homomorphisms [4]
	A Method for Obtaining Digital Signatures and Public-Key cryptosystems [34]

2.1.1 Historic Overview

The notation of homomorphic encryption has existed for decades, as Table 2.1 shows. Rivest *et al.* first proposed special encryption functions called *"privacy homomorphisms"* in 1978 [4]. They initially discussed the use of hardware to process data securely, where the data is only decrypted on a physically secure processor (discussed further in Section 2.3). Next, they describe a solution where data is encrypted while still allowing standard hardware to compute operations over it – the first notion of concept of homomorphic encryption. Proof-of-concept examples were given by Rivest *et al.* , showing that the concept of processing encrypted data could be possible, concluding with two open questions [4]:

Question 1 *Does this approach have enough utility to make it worthwhile in practice?*

Question 2 *For what algebraic systems does a useful privacy homomorphism exist?*

Later in 1978, the Rivest-Shamir-Adleman cryptosystem (RSA) was made public. The original paper had no mention of homomorphic encryption; however, it was later proved to support multiplication over encrypted data [46]. This is due to the mathematical properties of RSA, where raising a message to e in modulo n allows the two encrypted messages to be multiplied correctly since $m_0^e \times m_1^e \mod n \equiv (m_0 \times m_1)^e \mod n$. Therefore, partially homomorphic encryption has been conceptually supported since the idea of homomorphic encryption was conceived, though it was only realised years later [46].

Similar to work by Davida *et al.* [48] in 1981, Blakley *et al.* proposed a database encryption scheme, which supported computing statistical operations over the encrypted data [45]. Even though the paper did not specifically mention homomorphism, this was another step in proving that processing encrypted data could be made worthwhile in practice. The terminology used by Blakley *et al.* for protecting data was *hiding needles in a haystack* [45], where a large number of possible options exist, but only one is correct. The example provided required finding the Greatest Common Divisor (GCD) of approximately 2^{162} pairs [45]. It would be infeasible to try each pair, even

when performing millions of checks per second. This concept of *hiding in the masses* is one of the security properties that the work in this thesis builds upon.

In 1987, Ahituv *et al.* provides some example algorithms that can support homomorphic operations [44]; however these have very weak security. The following year, Brickell *et al.* evaluated the first known proposal of homomorphic algorithms by Rivest *et al.* [4], proposing an additive homomorphic algorithm where there was a maximum number of additions that could occur before it would break [43]. However, it was not until 1996 that the first homomorphic encryption scheme to support both addition and multiplication was proposed [42]. Ferrer provided a needed breakthrough in the field, and later showed in 2002 that the proposed scheme was secure against known clear-text attacks; however, the scheme was broken a year later by Wagner [49]. Armknecht *et al.* proposed a solution that supported an arbitrary number of addition operations, but only a fixed number of multiplications, while remaining secure under a known decoding problem [40].

Until this point, challenges that proposed schemes encountered were either: (1) not secure, (2) not able to maintain homomorphic properties, (3) not able to support repeated operations, or (4) incredibly inefficient. The breakthrough occurred in 2009 when Gentry proposed a scheme that could support an arbitrary number of additions and multiplications, and the security was based on the hardness of lattice problems [39]. Since then, Gentry *et al.* have been involved in proposing many schemes [7][8][9] for fully homomorphic encryption, gradually improving the schemes; however, a practical solution is yet to be discovered.

2.1.2 Overview of Partially Homomorphic Encryption Cryptosystems

This section will describe a few common partially homomorphic encryption cryptosystems: RSA, El Gamal, Paillier and Goldwasser–Micali. The encryption and decryption algorithms are given to prove their homomorphic properties. A similarity between the schemes is that each multiply cipher values within a large modulo to compute their homomorphic function. Hence this operation was the first to be analysed in this thesis for making homomorphic encryption practical, leading to Hypothesis 1 (discussed in Appendix A).

Rivest-Shamir-Adleman Cryptosystem

The encryption and decryption algorithms for Rivest-Shamir-Adleman cryptosystem (RSA) are now given: $E(m) = m^e \mod n$ and $D(c) = c^d \mod n$. Both raise a message (plain or cipher) to an exponent inside a modulo n. The proof of RSA supporting a homomorphic multiplication operation is given below. The two messages are multiplied together because their exponents are the same; therefore, two cipher values can compute a multiplication operation.

$$\begin{aligned} E(m_0) \times E(m_1) &= m_0^e \times m_1^e \mod n \\ &= (m_0 \times m_1)^e \mod n \\ &\therefore m_0 \times m_1 \mod n \end{aligned}$$

El Gamal

The security strength of El Gamal is based on the NP-hardness of solving discrete logarithms, which was first proposed in 1985 by Taher Elgamal [5]. There is no publicly known algorithm to solve (break) it quickly (assuming a large key size, for example 2048 bits). The encryption algorithm $E(m) = [\alpha^k \mod p,\ \beta^k \times m \mod p]$, where k is random and the remaining are public knowledge, is unique in that the cipher is made up of two parts. The decryption algorithm joins these parts together with the secret d, $D(c) = (\beta^k \times m) \times (\alpha^k)^{-d} \mod p$. The important aspect for this thesis is that El Gamal supports homomorphic multiplication operations on encrypted data, proven below. The two β values can be seen as the encryption of a single value because the ks are random, so they can just be combined into another random value.

$$\begin{aligned} E(m_0) \times E(m_1) &= [\alpha^{k_0} \times \alpha^{k_1} \mod p,\ \beta^{k_0} \times m_0 \times \beta^{k_1} \times m_1 \mod p] \\ &= [\alpha^{k_0+k_1} \mod p,\ \beta^{k_0+k_1} \times m_0 \times m_1 \mod p] \\ &= [\alpha^{k} \mod p,\ \beta^{k} \times m_0 \times m_1 \mod p] \\ &\therefore m_0 \times m_1 \mod p \end{aligned}$$

Paillier

Proposed in 1999 by Pascal Paillier [6], the Paillier cryptosystem is based on the fact that computing n^{th} residue classes is hard. The encryption algorithm is $E(m) = g^m \times r^n \mod n^2$, where r is random, and decryption is achieved through $D(c) = L(c^\lambda \mod n^2) \times u \mod n$. The homomorphic operation supported by the Paillier cryptosystem is addition. This is achieved by multiplying two cipher values together and is proven below. Addition is achieved because the message m is now the exponent, unlike RSA. The random values (r_0 and r_1) can just be combined as a single random value r.

$$\begin{aligned} E(m_0) \times E(m_1) &= (g^{m_0} \times {r_0}^n)(g^{m_1} \times {r_1}^n) \mod n^2 \\ &= g^{m_0+m_1} \times r_0^n \times r_1^n \mod n^2 \\ &= g^{m_0+m_1} \times r^n \mod n^2 \\ &\therefore m_0 + m_1 \mod n \end{aligned}$$

Goldwasser–Micali

The previous partially homomorphic encryption cryptosystems have encrypted integers, whereas the cryptosystem presented by Shafi Goldwasser and Silvio Micali in 1982 only encrypts individual bits [47]. This is because it is based on the quadratic residuosity problem, where a cipher value is either a quadratic residue or not for some modulo (only two possible results). The encryption algorithm, $E(m) = y^2 \times x^m \mod n$ where $m \in \{0, 1\}$ and y is random, such that $gcd(y, n) = 1$ or y is in the group of units modulo n. The decryption algorithm just performs the test to see if the cipher value is a quadratic residue in modulo n, resulting in two possible values $\{0, 1\}$. This cryptosystem is unique in that it supports a homomorphic XOR operation, as proven below. Technically, it is still an addition operation; however, the result can only be $\{0, 1\}$. Therefore, two bits of value 1 will make x^2, which is equivalent to x^0.

$$\begin{aligned} E(m_0) \times E(m_1) &= y_0^2 \times y_1^2 \times x^{m_0} \times x^{m_1} \mod n \\ &= y_0^2 \times y_1^2 \times x^{m_0+m_1} \mod n \\ &\therefore m_0 + m_1 \mod 2 \\ &\therefore m_0 \oplus m_1 \end{aligned}$$

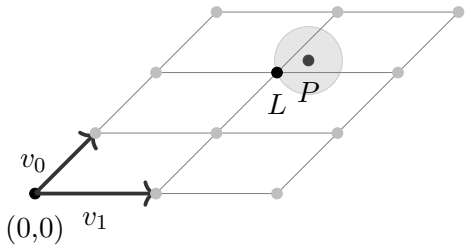

Figure 2.1: Finding the nearest lattice point for a point P in 2-dimensions

2.1.3 Lattice Based Cryptography

Fully homomorphic encryption schemes are primarily based on the hardness of lattice-based problems. Two lattice problems which cryptography can be built upon are the Closest Vector Problem (CVP) and Shortest Vector Problem (SVP) [50].

Definition *Given a point P and the lattice M, the Closest Vector Problem asks us to find the nearest lattice point L in M.*

In Figure 2.1, it is easy to see the closest lattice point in pictorial form, but because a computer is only given a matrix, and the dimensions are much larger, it makes finding the closest lattice point more computationally intensive.

Definition *Given the lattice M, the Shortest Vector Problem asks us to find the smallest non-zero vector V in M.*

Non-zero is important because every lattice technically has a zero vector. In Figure 2.1, this is simply v_0, because the example lattice is only showing the basis vectors. However, if larger vectors are given, there may exist a smaller vector which can make up the fundamental region. Note that CVP is actually a generalization of SVP, because if you are given an oracle (function) for CVP, it is possible to find the shortest vector by querying the oracle [51][52]. Therefore, because CVP is a NP-Hard problem, so is SVP [53][54].

2.1.4 Learning With Errors

The Learning With Errors (LWE) problem was introduced by Regev [55] and is widely used for a range of cryptographic functions, such as public key encryption [56]. This is because it claims to be as hard as worst-case lattice

problems [55], implying that any functions built upon LWE are secure. It is also currently assumed that worst-case lattice problems are even secure against the likes of quantum computers [57].

Definition *The Learning With Errors problem asks us to recover a secret* $x \in \mathbb{Z}_q^n$ *given a sequence of "approximate" random linear equations on* x *[55].*

Relating this back to lattices, the definition requires that distinguishing vectors are created from a set of noisy (contains some error) linear equations between uniformly random vectors. Looking at an example from Regev [55], we are given the set of linear equations shown below where in this case each equation has an error of approximately ± 1, and all we have to do is solve for x. However, by introducing the error it makes solving x more difficult. If we were to try and solve the set of equations using Gaussian elimination (row reduction) for example, the errors would accumulate, making the result invalid.

$$
\begin{aligned}
14x_1 + 15x_2 + 5x_3 + 2x_4 &\approx 8 (mod 17) \\
13x_1 + 14x_2 + 14x_3 + 6x_4 &\approx 16 (mod 17) \\
6x_1 + 10x_2 + 13x_3 + 1x_4 &\approx 3 (mod 17) \\
10x_1 + 4x_2 + 12x_3 + 16x_4 &\approx 12 (mod 17) \\
9x_1 + 5x_2 + 9x_3 + 6x_4 &\approx 9 (mod 17) \\
3x_1 + 6x_2 + 4x_3 + 5x_4 &\approx 16 (mod 17) \\
&\dots \\
6x_1 + 7x_2 + 16x_3 + 2x_4 &\approx 3 (mod 17)
\end{aligned}
$$

Given $2^{O(n \log n)}$ equations, we can deduce the secret x in $2^{O(n \log n)}$ time but this approach is based more around luck than sense. A simpler technique is the Maximum Likelihood algorithm, which only requires $O(n)$ equations and computes the only value for x that can satisfy the equations. This is achieved with brute force by computing all possible values, resulting in a runtime of $2^{O(n \log n)}$ [55]. Currently, the best-known algorithm for solving LWE is by Blum *et al.* and requires $2^{O(n)}$ equations and time [58], which relates to the fact that the best algorithms for solving lattice problems need $2^{O(n)}$ time [59][60].

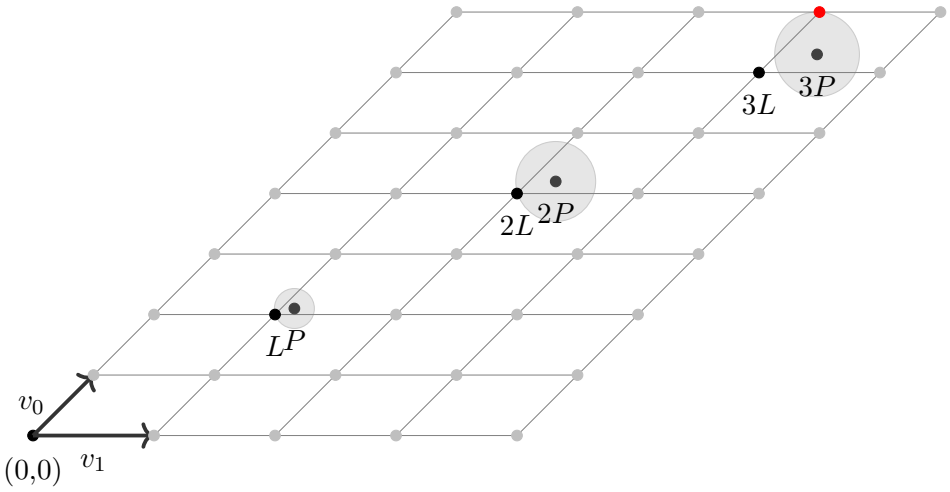

Figure 2.2: Combinations of errors or noise can lead to invalid results

Therefore, homomorphic schemes built on the LWE problem need to keep the errors relatively small, otherwise results can become invalid. For example, Figure 2.2 shows a lattice point L, and the point P which has some error added, so that by solving CVP, L will be the result. If P is doubled, $2L$ is still the closest lattice point; however, by adding P again giving $3P$, the closest lattice point is not $3L$, but instead $3L + v_0$, therefore the result is now invalid. Managing this error is one of the key challenges to homomorphic encryption in general and is a contributing factor to the performance limitations of current fully homomorphic encryption schemes, such as the Approximate Eigenvector algorithm.

2.1.5 Approximate Eigenvector Algorithm

The Approximate Eigenvector scheme for fully homomorphic encryption was proposed by Gentry *et al.* [9] in 2013, which uses the learning with errors problem described in the previous section (Section 2.1.4). This scheme allows for addition and multiplication homomorphic operations to be computed the same as matrix addition and multiplication operations, for most cases. Gentry *et al.* claim this makes the scheme asymptotically faster and easier to understand [9]. The formal definition of an eigenvector is given below, where T is a square matrix, and λ is a scale. Essentially for $\overrightarrow{v}$ to be classified as an eigenvector for the matrix T, the result must be equivalent to multiplying $\overrightarrow{v}$

by some scale factor. In this scheme $\overrightarrow{v}$ is an approximate eigenvector, not a perfect one.

Definition *A vector $\overrightarrow{v}$ is an eigenvector (also known as a characteristic vector, proper vector, or latent vector) if $T(\overrightarrow{v}) = \lambda\overrightarrow{v}$ for some scale λ [61].*

To make this a levelled scheme (i.e. it will work up to a certain depth), the error must be controlled. As seen in Figure 2.2, controlling the size of the error is extremely important, so that the decrypted value will be correct. This is achieved by flattening the cipher matrix so that it contains values in $\{0, 1\}$. Flattening uses bit decomposition operations, which does not affect dot product operations on the matrices. This allows homomorphic operations to still be computed correctly while keeping the errors small.

2.1.6 Practicality of Fully Homomorphic Encryption Schemes

Recent work is concluding that fully homomorphic encryption is still not efficient or practical [14]. For example, Gentry *et al.* showed in 2012 the performance for an Advanced Encryption Standard (AES) 128-bit circuit, which took 40 minutes per AES block [62]. Wang *et al.* [10] showed performance results of a revised fully homomorphic encryption scheme by Gentry and Halevi [63] in 2015 for the recrypt function. Central processing unit and graphics processing unit implementations took 17.8 seconds and 1.32 seconds respectively, using a small dimension size of 2048 [10]. A medium dimension size of 8192 took 96.3 seconds and 8.4 seconds for the same function [10]. With operations taking seconds, cloud implementations can become unusable. For example, adding millions of votes for a ballot requires weeks instead of hours. This is the challenge for current fully homomorphic encryption schemes, where recrypting the data to manage the error is computationally intensive.

Designed for *Negative-AND (NAND)* gates, while providing *XOR* gates for *"almost free"* – similar to functionality proposed in Part II – Ducas *et al.* present the Fastest Homomorphic Encryption in the West (FHEW) scheme which can bootstrap much faster than any other fully homomorphic encryption schemes [11]. This was based on work by Alperin-Sheriff *et al.* [64][65], while the performance was achieved with the fast Fourier transform and the highly optimised library: Fastest Fourier Transform in the West. Results given

showed the homomorphic NAND and bootstrap operation took 0.69 seconds, using 2.2 gigabytes of memory. Ducas *et al.* note that the scheme needs to be further optimised and generalised to reach its full potential. The current limitation of the scheme is the large cipher-text sizes for single bits and constructing an operation (addition, for example) from the NAND logic will require many bootstrap operations. FHEW is discussed further in Chapter 3, where it is practicality is compared to other schemes.

2.2 Secure Multiparty Computation

2.2.1 History

First proposed by Yao in 1982 as secure two-party computation, it aimed to answer the following question [15]:

Question *Two millionaires wish to know who is richer; however, they do not want to find out inadvertently any additional information about each other's wealth. How can they carry out such a conversation?*

The idea is such that two parties have an input they want to keep private from the other while computing some function. Yao demonstrated in 1986 how to generate a random integer using two-party computation, such that it remained secret from both parties [16], particularly useful for distributed key generation. It was later generalised in 1987 to *m-Party* cases, such as voting between m committee members where their vote is their private input [17]. This lead the way for work on fault tolerance and verifiability with majority parties being honest [66][67][68]. A brief overall history can be found in Table 2.2, with improvements for performance, and the addition of homomorphic encryption appearing, before focus on cloud implementations started to be published.

With the data being computed across different parties with frequent communication, there is a latency overhead penalty similar to the computational overhead of fully homomorphic encryption. This is why multi-party computation, even though has been around since 1982, still has not realised practical performance for the general use case. Improvements are occurring rapidly, where the performance of a circuit with thousands of logic gates went from minutes to seconds within a few years [69][70] using two-party computation.

However, at least three parties are desirable to handle malicious parties and improved sharing security. Given circuits often contain logic that is independent of one another during stages of computation, parallelisation is one technique for improving performance [71]. A technique called "cut-and-choose" (a two-party protocol) was used to provide greater security against malicious parties, by creating multiple garbled circuits [72][73], and is faster than using zero-knowledge proofs for verification. Given most schemes offer a free XOR function – no network transfer – practicality can be improved by reducing the number of non-XOR gates in a circuit [74]. This is a technique that can be used with the work presented in this thesis.

The reason circuits, or simply the LUTs cannot be reused is they allow patterns to be monitored, where the occurrences for the matching row can start to reveal what the input and output are. Work around reusing circuits in the cloud was presented by Wang *et al.* [75], using an *all-or-nothing* privacy approach. The client stores partial data in the cloud (single server), and submits some garbled inputs for processing, where if one garbled input is leaked, the other inputs are also leaked. The inputs could be encrypted, where the function first needs to decrypt the inputs before the actual function is computed. They state that if the cloud does not learn the result of the function, then it is the same as traditional semantic security. The limitations are that only some of the client data is stored in the cloud and that a single server evaluates the garbled circuit used for processing.

2.2.2 Garbled Circuits

In order to hide a party's input, obfuscation is used in the form of garbled circuits: defining logical circuits by creating LUTs with each input and output combination. An example is given in Table 2.3 for a NAND function. A party first generates the table and gives the inputs and outputs random labels (X), thus obfuscating them. Then the inputs are encrypted before the rows are shuffled, making the table seem garbled. Because of the shuffling, knowing the resulting row or output value does not reveal the input. The necessary labels and garbled circuit are then sent to the other party. This party uses its input to try and decrypt each row, where only one will decrypt giving the result,

Table 2.2: Historic overview of multi-party computation

2016	Garbled Computation In Cloud [75]
2013	Minilego: Efficient Secure Two-Party Computation from General Assumptions [73]
2012	A New Approach to Practical Active-Secure Two-Party Computation [70]
	On-the-Fly Multiparty Computation on the Cloud via Multikey Fully Homomorphic Encryption [20]
2009	Secure Two-Party Computation Is Practical [69]
2001	Multiparty Computation from Threshold Homomorphic Encryption [76]
1998	Efficient Multiparty Computations with Dishonest Minority [77]
1992	Communication Complexity of Secure Computation [78]
1989	Verifiable Secret Sharing and Multiparty Protocols with Honest Majority [68]
1988	Cryptographic Computation: Secure Fault-Tolerant Protocols and the Public-Key Model [66]
	Multiparty Unconditionally Secure Protocols [67]
1987	How to Play any Mental Game [17]
1986	How to Generate and Exchange Secrets [16]
1982	Protocols for Secure Computations [15]

Table 2.3: Garbled circuit truth table for a NAND gate

a	b	c
0	0	1
0	1	1
1	0	1
1	1	0

→

a	b	c
X_0^a	X_0^b	X_1^c
X_0^a	X_1^b	X_1^c
X_1^a	X_0^b	X_1^c
X_1^a	X_1^b	X_0^c

→

c
$Enc_{X_0^a X_0^b}(X_1^c)$
$Enc_{X_0^a X_1^b}(X_1^c)$
$Enc_{X_1^a X_0^b}(X_1^c)$
$Enc_{X_1^a X_1^b}(X_0^c)$

→

c
$Enc_{X_1^a X_0^b}(X_1^c)$
$Enc_{X_0^a X_1^b}(X_1^c)$
$Enc_{X_1^a X_1^b}(X_0^c)$
$Enc_{X_0^a X_0^b}(X_1^c)$

which is then shared. A few challenges faced by multi-party computation are highlighted in this example, which this thesis aims to solve:

- Because the output is typically known to all parties, this could be seen as problematic when the result must remain private from the cloud providers (the output should only be visible to the client or end-user).
- Clients are transferring garbled circuits, which is a large amount of data for a simple logic gate.
- The circuit cannot be reused, as this will reveal patterns. Recent work has started to address this [79][80][81].
- Each row must be tried to find the result, slowing down the lookup times.
- The circuit is generally evaluated on one server.

2.2.3 Secret Sharing

Widely used in multi-party computation schemes, secret sharing is the act of distributing a piece of information across many parties, where $k \leq n$ pieces are then required to recover the original value. Therefore, $< k$ pieces cannot reveal the value, keeping it secret [82]. If $k < n$, then the scheme is known as a threshold scheme; these were first proposed in 1979 by Blakley [83] and Shamir [84]. Threshold cryptography is an example of secret sharing where several parties or servers are needed to decrypt the value. The security strength of this approach comes from the fact that k servers need to be compromised. The same approach is used in this thesis, where multiple servers protect data privacy.

Secret sharing exists in two forms: secure and insecure. The difference is that for a sharing technique to be secure, as more pieces are obtained, the number of possibilities or complexity of solving the original value should not change. For example, when splitting the phrase "secret" into "se", "cr" and "et", if two pieces are obtained, the number of possibilities for the result word decreases. However, if the secret is converted into 0x736563726574 and split into 0xC0E300001752 + 0xE2F77285900D + 0xCF8AF0ECBE13 mod 0xFFFFFFFFFFFF, then the number of possibilities does not change with two pieces.

Shamir Secret Sharing

The basic concept of Shamir secret sharing is that there is a minimum number of points k to define any polynomial [84]. In particular, there are k points needed to define a polynomial of $k-1$ degree. For example, there are 2 points required to define a line, 3 points to define a parabola, and 4 points to define a cubic curve. Therefore, on secret generation, n points are created on the polynomial, but only k are required to define the polynomial and find the secret.

Blakley Secret Sharing

Blakley's secret sharing scheme is based on the principle that any N nonparallel $(N-1)$th dimensional hyperplanes intersect at a specific point [83]. The shares are hyperplanes, where the secret is the intersection.

2.2.4 Combining Multi-Party Computation and Homomorphic Encryption

With multi-party computation, the computation of a function over some data is secure – much the same as homomorphic encryption, but if secret sharing is not used, all parties have visible private inputs. With the parties and their data hosted in the cloud, the *"private"* inputs are no longer protected. Protecting the private inputs can be achieved with homomorphic encryption, using either a shared key or multiple keys for each party. Shared keys are self-explanatory, where each input is encrypted with the same key. The more interesting situation is where inputs are encrypted with different keys.

López-Alt *et al.* implemented *on-the-fly multi-party computation* in the cloud, while using homomorphic encryption to encrypt inputs using multiple, unrelated keys [20]. Using a multi-key lattice fully homomorphic encryption scheme [18][19], each user can encrypt their data with their key and send it to the cloud, where functions can be applied over many users' encrypted data (for example, computing the average value). The result of any secure computation is decrypted by all the keys that were used as inputs. The benefit over more traditional multi-party computation is that the private inputs are protected, thus ideal for a cloud environment. The downside of such a scheme is that by using fully homomorphic encryption, the overall computational performance is still poor.

2.2.5 Current Libraries and Implementations

VIFF

The Virtual Ideal Functionality Framework (VIFF) was developed in Python for secure multi-party computation, with the goal to facilitate rapid prototyping, leading to practical multi-party computation applications [85][86].

SEPIA

SEcurity through Private Information Aggregation (SEPIA) [87][88] is a Java library proposed for generic secure multi-party computation. The original use case was for collaborative network monitoring to detect events and calculate statistics. This allows different domains, networks, and organisations to work together to detect intrusion alerts while keeping their traffic and alerts hidden from the other peers.

FairplayMP

Initially proposed in 2004 as Fairplay [89] for two-party computation, FairplayMP [90][91] is a system that supports secure multi-party computation. It constructs boolean gate tables, from a high-level function definition, automating the circuit creation.

SPDZ and SCALE-MAMBA

SCALE-MAMBA [92] was born from an early version of SPDZ [93][94], with the goal of creating a production ready system. Both have good support for compiling Python programs, but the offline generation stage is the main bottleneck, for example, generating the triples for multiplication (a principle proposed by Beaver [95]).

TinyGarble

A logic synthesiser and circuit optimiser, TinyGarble was proposed in 2015 to minimise the number of non-XOR gates for custom hardware [74]. Songhori *et al.* state that this makes garbled circuits more practical, but the issues of data transfer still exist, and deployment in the cloud is difficult.

Sharemind

Sharemind is a framework for independent organisations to collaborate and process information in a privacy-preserving manner [96][97]. The importance of Sharemind is not new technologies, because it is built upon existing secure multi-party computation protocols, but the business model of using independent organisations. This model provides trust by each processing server being managed by a different organisation, where five are currently available. The most efficient setting for Sharemind is with three servers, where one can be semi-honest [96]. However, the trust model breaks when multiple organisations act maliciously and communicate with each other to attempt to reveal some data. An interesting use case for Sharemind is estimating satellite collision probabilities in low orbit [98], supported by the US Defense Advanced Research Projects Agency. Limitations for certain applications are that the closed-source design inhibits comprehension and auditing [99].

Lightweight Web Application

Recent work implemented a multi-party computation scheme and web application for the Boston Women's Workforce Council with regards to a study on wage disparities [99][100][101]. Lapets *et al.* also introduce the concept

of *multi-party computation-as-a-service* [99]. The operation required was an aggregate statistic (sum) over the data with regards to gender and job category. Given the single operation, this application is not comparable to fully homomorphic encryption, as arbitrary computation is not supported. However, it does show how certain applications or use-cases can benefit different privacy-preserving schemes.

2.3 Secure Hardware Processors

Another method and the earliest form of practical privacy-preserving computation is developing custom hardware processors, also mentioned by Rivest *et al.* [4] in 1978. An example is the state-of-the-art secure processor AEGIS [21]; designed to only reveal the data inside the processor, any data entering or leaving the processor is encrypted, for example to external memory. This protects against a range of software and physical attacks. However, AEGIS still has security vulnerabilities in the form of side-channel-attacks [102][103]. This attack vector analyses information "leaked" from the physical execution of a program, for example power consumption [104] or electromagnetic radiation [105]. Other limitations of secure processors are the practicality of deployment in the cloud, because they are a physical entity.

By creating services reliant on custom hardware, we lose the core essence of what the cloud should be, which is abstract and dynamic [36]. However, users can now deploy Field-Programmable Gate Array (FPGA) designs in the cloud for custom hardware accelerators [106]. A FPGA bridges the gap between hardware and software, by providing performance closer to that of an application-specific integrated circuit, while having the reconfigurability of a microprocessor. They contain a finite amount of programmable logic, also known as reconfigurable logic, that can be used to implement digital circuits by applying a bitstream file to the device. This bitstream file is analogous to a compiled program in software, but where programs contain machine instructions, a bitstream file contains a sequence of bits which configure circuits and logical functions. Processing data in parallel can give better performance; however, it is usually achieved by computing the same function over chunks of data at the same time. FPGAs offer a slightly different form of parallel

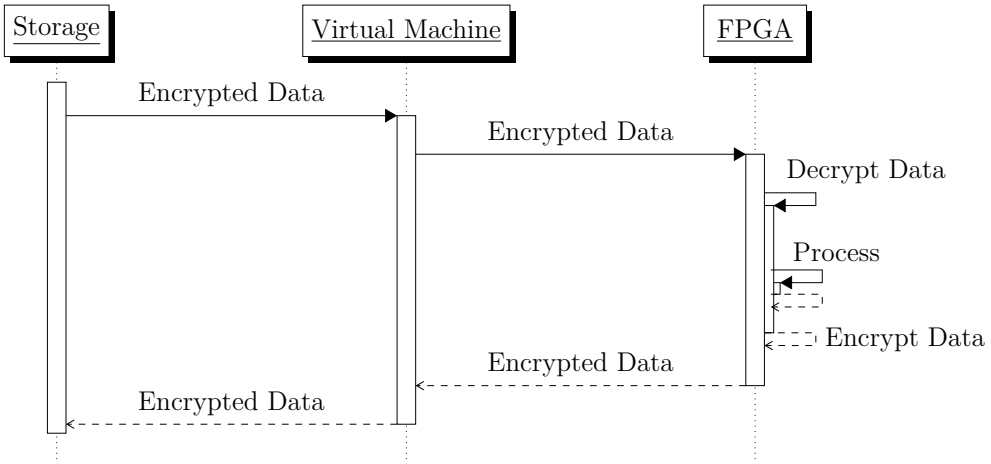

Figure 2.3: SFaaS data flow for secure processing

processing by pipelining a design. This allows different stages of a function to be processed in parallel such that the data flows through the function, giving the performance that is currently lacking with secure processing. The design can either be application specific or based on an instruction set processor [107].

Using a FPGA in the cloud could allow a data flow process similar to Figure 2.3 [36], where the virtual machine acts as the controller. The bitstream file is encrypted with the FPGAs public key and includes the decryption key for the data. This allows symmetrical keys to be used for data processing, giving greater performance and using less logic over a public key cryptosystem. Providing the decryption key for the data stops malicious users of the FPGA being able to decrypt the data by writing a bitstream file to decrypt other users' data.

2.3.1 Physical Unclonable Function

A Physical Unclonable Function (PUF) allows for secrets to be derived from complex physical characteristics of the silicon (a physical one-way function) rather than storing the secrets in memory [108][109][110]. Guajardo *et al.* proposed the feasibility of using PUFs for intellectual property protection by encrypting the bitstream file using elliptic curve cryptography [111]. PUFs are naturally noisy, which combined with varying temperatures and ageing could affect the reliability [112]. However, the key only needs to be generated or regenerated once per power cycle. Therefore, after the FPGA is powered on

and the private key has passed a test against the known public key, any PUF issues will not be encountered until it is reset. Even though the feasibility of PUFs is still to be discovered, the idea of having a decryption key only visible to the FPGA could be realised soon.

The alternative to using a PUF is where a FPGA is hardcoded with a private key which is generated automatically during manufacturing, and the public key is printed on the chip or included in the box. The keys should never be saved, and only visible for a brief moment during manufacturing. The manufacturer could also sign all public keys, allowing customers to verify the key was generated by the manufacturer, not by a malicious entity. However, the private key is visible at some point in time, so is not the ideal solution.

2.3.2 Programmed Decryption Key

With a protected bitstream file, the decryption key for data processing can remain protected, as it will only be exposed within the FPGA. Three cryptography schemes will be analysed for usage in an FPGA in terms of logic size and performance. Note that logic size depends on the FPGA used, and is given as an approximation.

Advanced Encryption Standard

Rijndael, or as it is commonly known, AES, is a method of data encryption with symmetric keys [113]. The advantage over other cryptography schemes for layered encryption is the cipher-text is the same size as the plain text. For example, with a 2048-bit RSA encryption key, a 32-bit plain-text value increases by a factor of 64. Adding another layer of encryption grows the size again, as a 2048-bit value needs to be split into smaller chunks to be encrypted with another 2048-bit key. Therefore, for layered encryption, the property of fixed sized cipher text is important. With inputs larger than the key size, they are split into blocks and chained together. The core operations of AES are XOR and bit rotating, giving it good performance as well as small cipher text sizes. The two components required for encryption and decryption are the key (256-bit for example), and initialisation vector.

The AES is widely used in secure FPGA designs and offers very fast per-

formance (> 20Gbps) [114][115][116]. Designs can also be tailored to use less logic, from thousands of slices down to a few hundred while still achieving megabits per second [115][116]. The advantage of AES over other cryptography schemes in terms of implementation on an FPGA is the simplicity of the algorithm: the flexible performance versus logic required. The only main limitation of AES is that it uses a symmetric key. Even with the key protected in the bitstream file, data sharing and multiple data sources remain an issue. However, AES could be used for data storage and processing, while all uploaded or outgoing data could be encrypted with a public key cryptosystem.

Rivest-Shamir-Adleman Cryptosystem

Recall that Rivest-Shamir-Adleman cryptosystem (RSA) is a public key encryption scheme designed on the factoring problem, computing the plain-text or cipher base value to an exponent within a modulo [34], and can use Montgomery modular multiplication [117][118]. RSA is more expensive in terms of area and performance compared to AES for FPGAs. For example, a slower implementation still requires thousands of slices, where faster implementations can require tens of thousands [119]. Performance can vary between megabits per second down to kilobits per second [118].

Elliptic Curve Cryptography

A public key cryptosystem, Elliptic Curve Cryptography (ECC), was proposed independently by Neal Koblitz and Victor Miller in 1985, and its cryptographic strength comes from the elliptic curve discrete logarithm problem being hard [120]. The advantages of ECC are smaller key sizes, smaller cipher text sizes (less data transferred) and faster computation times [121][122] when compared with non-ECC schemes such as RSA. For example, the *secp256r1/nistp256* curve (256-bit) is comparable to the cryptographic strength of an RSA 3072-bit key [123]. An early FPGA implementation by Leung *et al.* showed a $\times 30$ speedup over software implementations, using only a few thousand slices [124]. A recent survey in 2007 showed the varying difference between state-of-the-art implementations [125], and in 2008, a 33.05μs solution on a 163 bit binary field was proposed [126].

2.3.3 Attack Vectors

Apart from black box attacks, all other attack vectors require physical access, including readback attacks, side-channel attacks and reverse engineering an FPGA bitstream file [127]. This is an improvement over current software solutions for data processing, which can be accessed and broken remotely.

Black Box Attacks

These are a common attack for systems where all possible input combinations are tried, with the output revealing the inner design [127]. This attack is not feasible given that the input and output data must be encrypted, where the public key may not even be known. The design should handle incorrect input, for example a value not encrypted with the correct public key.

Readback Attacks

For debugging, FPGAs often have a readback feature to allow values to be read from the FPGA, for example the decryption keys through a special interface. Methods of disabling this functionality exist [127]; however, for a production FPGA deployed in a cloud service, this functionality should not exist or be physically disabled in the chip once the chip has passed production tests.

Side-Channel Attacks

These attack vectors are viable and involve analysing physical properties of the FPGA while in operation, for example power consumption [104] or electromagnetic radiation [105]. These are difficult and can require a laboratory environment to be successful.

Reverse Engineering the Bitstream File

With enough time and effort, the design of a bitstream file (once decrypted) can be reverse engineered [127]. However, an attacker only need focus on finding the decryption key for the input data, which could be easier depending on the level of obfuscation.

2.4 Secure String Searching

A subset of secure data processing, string or query searching over encrypted data is a widely-researched domain, with many feasible solutions existing. Even though data is not technically processed such that it changes form, string searching is an important aspect of data storage in the cloud and was the first test case for Privacy-Preserving Encoding presented in this thesis.

Trapdoor functions, which are simple to compute in one direction but hard to reverse, have been used in cryptography for decades [34][35], and constitute the primary means of encrypted keyword searching [128][129][130][131][132][133][134]. Boneh *et al.* [128] describe one of the first applications for encrypted searching, a technique for associating a set of keywords with encrypted emails. This is achieved by the user creating a trapdoor for a word such as "urgent" using their private key, which the server uses to search the encrypted emails and inform the receiver when an email is urgent.

A major limitation of keyword indexes is that searches are only successful if their words are correctly spelt. Google's well-known example of 600 recorded versions of the query "Britney Spears" [135] is a dramatic illustration of wide variations in spelling. Today's users are accustomed to systems that are robust to spelling errors, and much research is aimed at supporting approximate searching in encrypted data [131][134][136][137]. Recently, schemes have been proposed to enable multi-keyword searching [138][139]. Li *et al.* show how to search over encrypted data while providing ranked results, a major improvement over previous schemes, but this requires the index to be built offline, with a trapdoor function for each word. A recent scheme for a searchable symmetric encryption protocol tried to find the balance between privacy and performance with regards to what can be leaked during a search [140]. This work highlights the need to look at alternate solutions which may not protect full privacy (access patterns are leaked) but can give practical usage.

For large enterprises, it is sufficient to build and maintain indexes within the corporate environment and transmit them to the cloud for use. However, ordinary people who use the cloud to store personal documents would probably prefer to delegate the task of building indexes and handling the searching to the service provider. Current systems either require the index to be downloaded,

updated, and re-uploaded or require heavy preprocessing on the document before being uploaded. Moreover, they do not support phrase or proximity searching, and at best are robust only to simple (single-character) typographical and spelling errors. Little research has been devoted to addressing these shortcomings.

2.5 Privacy

This thesis focuses on keeping data private from cloud providers (for example, the compute nodes), where information learnt from running operations over the data and seeing the result is out-of-scope. We have published in areas of privacy such as the location of requests [141], and user account information [142]. However, some other privacy techniques for data processing will be briefly mentioned here to further explore the privacy field.

In 2003, Dinur *et al.* showed that allowing any query, no matter how simple, over a private database could allow the leakage of an individual's privacy [143]. A solution to this problem is differential privacy, which aims to add noise to a statistical query result [12][13]. The objective is that the result from the query should be similar if some users data exists or not in the database [12][13]. By adding noise to the result means there is a tradeoff between privacy and accuracy, similar to the work presented in Chapter 4. The primary contribution of this thesis in Chapter 5 required accurate results but would allow techniques such as differential privacy to be applied as an additional layer.

Protecting user privacy within published data sets, k-anonymity states that a user cannot be distinguished from at least $k-1$ users in the same set [144][145]. Suggested as an improvement over k-anonymity, ℓ-diversity hardens against some of the known attack vectors on k-anonymity [146]. These techniques can be applied to data sets released by trustworthy vendors, such as healthcare releasing data sets for researchers. If the data set is published with these anonymity techniques already applied, then the work presented in this thesis would function as if the data set contained the real values. Thus, allowing the data to be fully private from the cloud providers, while the results when revealed would have a form of protection.

2.6 Summary

The approach that gives the greatest performance for privacy-preserving processing is custom hardware implementations, but as this chapter has highlighted, these do not fit the philosophy of the cloud. They also do not protect data for its entire life in the cloud and have other management issues [36], w whereas the solution that can continuously protect data – fully homomorphic encryption – is currently impractical to use in the cloud for generic applications, even with hardware implementations [147]. Partially homomorphic encryption has many variants and given the simplicity of the encryption algorithms can offer usability performance. However, Chapter 3 will present an application where usability performance is not guaranteed, including client-side performance. The limitation of a single operation also means partially homomorphic encryption cannot be the generic solution for privacy-preserving processing in the cloud.

With multi-party computation, the execution of a function over some data is secure – much the same as homomorphic encryption, but traditional definitions of multi-party computation state that all parties have private inputs. However, when the parties and their data are hosted in the cloud, the *"private"* inputs are no longer protected if they are plain-text values. Also, for any new party, they must also join the system, decreasing performance. Multi-party computation can be combined with fully homomorphic encryption to address this issue; however, then it has the limitations and concerns related to fully homomorphic encryption. Improvements to multi-party computation with garbled circuits still have a single server evaluating the circuit, and reusability of these circuits is still an open issue for data fully stored in the cloud. Work presented in this thesis aims at addressing these issues without the need for input data to be encrypted, answering Hypothesis 3.

Part I

Practicality of Privacy-Preserving Encoding

3

Defining Practicality

3.1 Introduction

Even when realising practical privacy-preserving processing, it will not be as efficient as processing plain-text values with current technology. Therefore, a benchmark is required to define what is practical in order to answer Hypothesis 3, and to be a target for the scheme presented in Part II. This chapter will concisely compare the performance of partially homomorphic encryption, fully homomorphic encryption, and multi-party computation for the use case of secure cloud voting.

Developing an electronic voting scheme which is 100% secure against all attack vectors is incredibly difficult, but it does need to be as secure as current paper voting techniques. To protect voter privacy, votes must be kept private so that no entity has knowledge of how someone voted. This is a perfect use case for privacy-preserving processing schemes because the result of a ballot is the sum of all votes for the available choices or candidates. Therefore, only a simple operation is required, meaning partially homomorphic encryption, fully homomorphic encryption, and multi-party computation can all be compared. There are many requirements an electronic voting scheme must meet [29]; however, for this example only the need to verify a vote as either a *yes* or *no* is required.

3.2 Comparing Schemes

Each scheme of the following schemes–plain text, partially homomorphic encryption, fully homomorphic encryption and multi-party computation–will be implemented in the simplest manner to meet this chapter's requirements for secure electronic voting. In this section *yes* or *no* with be represented as 1 or 0 respectively. Note that unless otherwise stated, all performance results in this chapter were computed on a MacBook Pro (Retina, 13-inch, Early 2013), and all results should be considered estimates or guidelines.[1]

3.2.1 Plain Text

Even though the performance of plain text processing is unfeasible for any privacy-preserving computation scheme to compete with, the performance of decrypting, processing, and encrypting the data can be compared. By encrypting the tally after each vote, it increases the chances the tally value is only visible within the central processing unit.

A simple example was created in Python using RSA, as shown in Listing 3.1.[2] Lines 2-5 are just for sample purposes, where lines 6-10 were profiled. Line 7 uses modulo 2 to guarantee the vote $\in \{0, 1\}$ while also providing randomness to the cipher votes. Line 10 attempts to clear the variables of their plain-text values. For 2048-bit keys, lines 6-10 took approximately 5ms, where 4096-bit keys took just under 30ms to compute. This is because of the decryption steps in lines 6 and 7. Even though the encryption and decryption algorithms for RSA are essentially identical, with encryption, a 32-bit number is raised to a large power but with decryption, a large number is raised to a large power. Therefore, even plain text processing still has computational overheads if the data is encrypted after each vote is added.

Listing 3.1: Simple Python voting example

```
1  from Crypto.PublicKey import RSA
2  tally_key = RSA.generate(2048)
3  vote_key = RSA.generate(2048)
```

[1] The laptop was in the same state for each experiment, and only the base operating system processes were running.

[2] Note that the value $11L$ is a random value such that modulo 2 reveals the vote (another form of randomness), while the tally is currently 123.

```
 4  vote_cipher = vote_key.publickey().encrypt(11L,None)
 5  tally_cipher = tally_key.publickey().encrypt(123L,None)
 6  tally =  tally_key.decrypt(tally_cipher)
 7  vote = vote_key.decrypt(vote_cipher) % 2
 8  tally += vote
 9  tally_cipher = tally_key.publickey().encrypt(tally,None)
10  tally = vote = 0
```

3.2.2 Partially Homomorphic Encryption

With the need to tally votes an additional homomorphic scheme is required. A C++ homomorphic library, *libhcs*, developed at the University of Waikato by Marc Tiehuis, was used for this example [29][148]. This library implements the threshold variant of the Paillier cryptosystem, and the library was primarily designed for a voting system [29]. The important property of this scheme is the ability to generate zero-knowledge proofs, in particular, proof of 1-out-of-2 n'th powers, for a given cipher value. These proofs allow the cloud service to verify a cipher vote contains either a 0 or 1, but it does not know which. Without this check, a malicious voter could encrypt large values to give their vote a higher weighting.

The implementation with *libhcs* is more complicated than other schemes presented in this chapter, with some small customisations to the library itself. The library was designed to distribute the voting, but only one server was required for this example. A generator value of 97 was used for the proofs, where a *yes* vote is the encrypted value of 97, and the *no* vote is 1. In Listing 3.2, line 8 sets the current tally value to 984. Not shown in this example is a counter to record the number of votes added, but for this tally, there have been 24 votes. Therefore, $984 - 24 = 960$ which when divided by the generator minus 1, gives 10 *yes* votes. Lines 10-12 encrypt a *no* vote, where the random number *ran* is needed later to generate the proof. Lines 13 and 14 generate a proof for the cipher vote, where the proof function also requires knowledge of the vote value. Lines 15 and 16 would occur in the cloud, where if the proof is valid, the cipher vote is added to the tally. The decryption would occur after the ballot has been completed and where the cloud should not have access to the decryption key.

Performance results for each core function are given in Table 3.1 using a 2048-bit key. The key generation is not critical for the running time as this is a one-off cost before the ballot begins. The total time for a client to generate

Table 3.1: Libhcs function execution times

Function	**Time (ms)**
KeyGen	140.3
Encrypt	14.5
ProofGen	32.6
ProofVerify	33.3
Decrypt	4.5
HomAdd	< 1

their cipher vote and proof would be $\approx$ 47.1ms. In the cloud, verifying each vote takes $\approx$ 33.4ms, and near no time to homomorphically add the vote to the tally. With these results, and without taking advantage of parallelisation, the cloud could process millions of votes in a single day. With cloud grade hardware (faster than a laptop) and parallel tallies, support for national elections is achievable ($2^{24} - 1$ votes were assume in Section 7.2.1).

To compare with Section 3.2.1, multiplying two large numbers within a modulo was tested in Python as well. Listing 3.3 shows the example code, where Line 6 performs the homomorphic addition. This line takes $<$ 1ms to compute, which is actually faster than the time in Section 3.2.1. Therefore, this example shows the overhead of decrypting and encrypting values each time a vote is added to the tally; where if a proof was not required, partially homomorphic encryption would actually be faster than computing on plain text for a simple addition.

Listing 3.2: Simple partially homomorphic encryption voting example

```
1  mpz_t tmp, cipher_vote, cipher_tally, ran;
2  mpz_init(tmp); mpz_init(cipher_vote);
3  mpz_init(cipher_tally); mpz_init(ran);
4  hcs_rand *hr = hcs_init_rand();
5  pcs_public_key *pk = pcs_init_public_key();
6  pcs_private_key *vk = pcs_init_private_key();
7  pcs_generate_key_pair(pk, vk, hr, KEY_SIZE);
8  mpz_set_ui(tmp, 984);
9  pcs_encrypt(pk, hr, cipher_tally, tmp);
10 mpz_set_ui(tmp, 1);
11 mpz_random_in_mult_group(ran, hr->rstate, pk->n);
12 pcs_encrypt_r(pk, cipher_vote, tmp, ran);
13 pcs_t_proof *pf = pcs_t_init_proof();
14 pcs_compute_1of2_ns_protocol(pk, hr, pf, cipher_vote, ran,
       0, 0);
15 if(pcs_verify_1of2_ns_protocol(pk, pf, cipher_vote, 0))
16         pcs_ee_add(pk, cipher_tally, cipher_tally,
               cipher_vote);
17 pcs_decrypt(vk, tmp, cipher_tally);
```

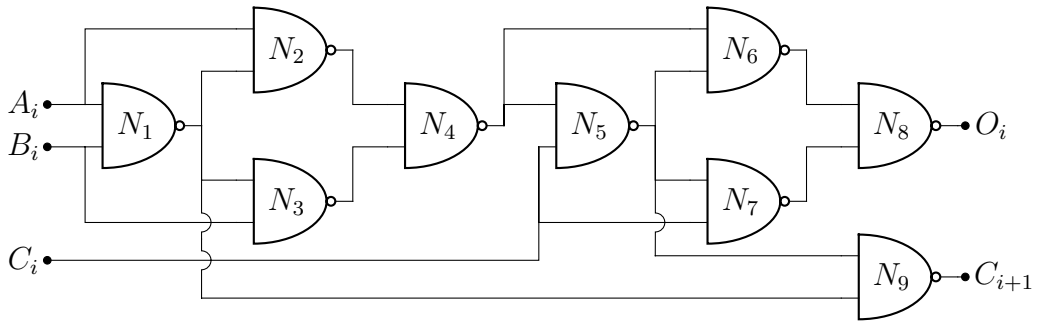

Figure 3.1: NAND gate full adder

Listing 3.3: Python multiplying large numbers example

```
1 import random
2 from Crypto.Util import number
3 n_sq = number.getPrime(2048)**2
4 r = lambda: random.getrandbits(4096)
5 c1, c2 = r(), r()
6 c3 = (c1 * c2) % n_sq
```

3.2.3 Fully Homomorphic Encryption

The performance of partially homomorphic encryption is manageable for cloud voting, although it does not have the ability to compute other operations: for example, being able to track which users have voted while keeping records private until after the ballot. More advanced ballots could be implemented, such as the mixed-member proportional voting method used in New Zealand. Therefore, ideally a fully homomorphic encryption scheme would be used to implement the voting system.

The library FHEW was used to implement a voting protocol, as discussed in Section 2.1.6, which provides the homomorphic NAND operation [149]. The addition of two 32-bit integers can be achieved with 31 full-adders and a single half-adder. A full-adder comprised of NAND gates can be seen in Figure 6.1. However, with a voting system, only a single bit will be received and added with the tally value. This also allows the scheme to guarantee a vote $\in \{0, 1\}$. In Figure 3.2, a half-adder is given, where from $i = 1$ the carry is the B input wire. Therefore, B_0 is the vote and A_i is the tally value. The number of total possible votes in the ballot, will vary the number of half-adders required.

Instead of implementing the entire voting protocol, only the performance of the required functions was evaluated. A sample program is shown in Listing 3.4

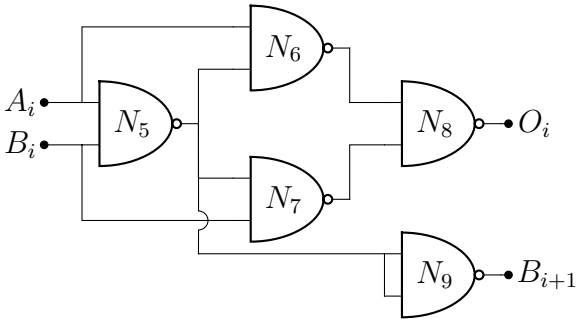

Figure 3.2: NAND gate half adder

Table 3.2: FHEW function execution times

Function	Time (seconds)
Setup	0.537
KeyGen(1)	< 0.001
KeyGen(2)	12.644
Encrypt	< 0.001
Decrypt	< 0.001
HomNAND	0.106

which gives the offline functions, encryption/decryption, and the homomorphic function.

Listing 3.4: Sample FHEW program for evaluation

```
1  int main() {
2          LWE::CipherText e1, e2, e12;
3          FHEW::Setup();
4          LWE::SecretKey LWEsk;
5          LWE::KeyGen(LWEsk);
6          FHEW::EvalKey EK;
7          FHEW::KeyGen(&EK, LWEsk);
8          LWE::Encrypt(&e1, LWEsk, 0);
9          LWE::Encrypt(&e2, LWEsk, 0);
10         FHEW::HomNAND(&e12, EK, e1, e2);
11         LWE::Decrypt(LWEsk, e12)
12         return 0;
13 }
```

The performance of each function was measured using the built-in clock function (processor clock ticks), and is shown in Table 3.2. Like the partially homomorphic encryption example, the key generation times are not critical, as they are performed offline. Given the scheme only encrypts or decrypts single bits, the performance for both functions is very fast. However, where this scheme and other fully homomorphic encryption schemes are inadequate when

computing a homomorphic function. In this case, computing a single NAND operation takes $\approx$ 100ms. Therefore, even with parallelisation, a half adder will take over 300ms to compute. If a 32-bit tally is required, 10s would be required to add a single vote to the tally. This can be improved by using a pool of tallies, each with a smaller number of bits, where 8-bits would still require a few seconds. To reach the same performance of the partially homomorphic encryption example voting scheme, nearly 100 parallel tallies would need to be computing at the same time. Note that this before any parallelisation on the partially homomorphic encryption voting scheme.

3.2.4 Multi-Party Computation

A use case for multi-party computation could be voting with a small number of fixed participants. However, with large-scale voting, an unknown number of completed votes will be added to the tally. The challenge with multi-party computation protocols is adding dynamic behaviour, as the protocols are constructed offline and agreed upon beforehand, where garbled circuits should not be reused. To compare multi-party computation, a simple tool was used to define a basic voting protocol.

TASTY is a Python tool developed for automating efficient secure two-party computation protocols using combinations of garbled circuits and homomorphic encryption techniques [150][151]. Because both partially homomorphic encryption and fully homomorphic encryption are already being evaluated, we will just use the garbled circuit methods for two-party computation. TASTY can also perform analysis and benchmarking for protocol designs, which are used below. The configuration used for each protocol in TASTY had a *long* security level, which gives: symmetric/statistical as 128-bits, asymmetric as $3,248$-bits, and curved as *secp256r1*. Both servers were on the localhost, with network traffic going through a pipe with a delay of 25ms in each direction.[3]

The first protocol design is given in Listing 3.5 where two servers (server and client in TASTY) add their values together. In terms of voting, the vote is split across the two servers such that $client_{val} + server_{val} = v$ where $v \in \{0, 1\}$;

[3]This latency value was chosen by measuring the latency between nine AWS locations, where 22.5ms was the average latency to each locations nearest neighbour (rounded up to 25ms).

in this case $3 + (-2) = 1$.

Listing 3.5: Simple TASTY voting design

```
def protocol(client, server, params):
    LEN = 32
    client.val = Signed(bitlen=LEN, val=3)
    server.val = Signed(bitlen=LEN, val=-2)
    client.gval = Garbled(bitlen=LEN, val=client.val)
    server.gval = Garbled(bitlen=LEN, val=server.val)
    client.ogval <<= server.gval
    client.gresult = client.ogval + client.gval
    client.result = Signed(bitlen=LEN+1, val=client.gresult)
    client.output(client.result)
```

The values are then garbled, and the server sends the garbled value to the client, which adds them and produces the result. To make this scheme comparable to the homomorphic encryption designs, there are a few elements missing. The first is that the result is not secured; to achieve this a random value r would need to be generated between the two servers where r is not known to either. Each server could sum the votes then perform an addition between them, but this is not really comparable. Also, the key feature missing is that the vote is not verified to be *yes* or *no*, which would require more computation than adding values. Listing 3.6 shows how multiple values were tallied together in this example protocol, where only two values are added together at one time.

Listing 3.6: Simple TASTY voting design for multiple votes

```
def protocol(client, server, params):
    LEN = 32
    client.val1 = Signed(bitlen=LEN, val=3)
        ...
    server.gval4 = Garbled(bitlen=LEN, val=server.val4)
    client.ogval1 <<= server.gval1
    client.ogval2 <<= server.gval2
    client.ogval3 <<= server.gval3
    client.ogval4 <<= server.gval4
    client.gresult1 = client.ogval1 + client.gval1
    client.gresult2 = client.ogval2 + client.gval2
    client.gresult3 = client.ogval3 + client.gval3
    client.gresult4 = client.ogval4 + client.gval4
    client.gresult12 = client.gresult1 + client.gresult2
    client.gresult34 = client.gresult3 + client.gresult4
    client.gresult = client.gresult12 + client.gresult34
    client.result = Signed(bitlen=LEN+6, val=client.gresult)
    client.output(client.result)
```

Protocols were generated for tallying N votes, where each vote is split across the two servers. The metrics given in Table 3.3 are overall times for the setup

Table 3.3: Multi-party computation results for a simple voting protocol

N	Setup (ms)	Online (ms)	Setup C->S (KB)	Setup S->C (KB)	Online C->S (KB)	Online S->C (KB)
1	457.09	55.08	3.38	4.89	0.12	1.56
2	856.34	157.78	6.67	10.93	0.24	3.11
3	1256.77	211.48	9.94	17.03	0.37	4.69
4	1374.16	318.10	8.06	27.56	0.49	6.25
5	1540.59	374.13	8.13	32.51	0.61	7.82
6	1798.90	484.97	8.22	37.29	0.73	9.39

and online phases, as well as the amount of data transferred during each phase. For some protocols, we could ignore the setup phase; however, if a verification protocol was designed to check that $(client_{val} + server_{val}) \in \{0, 1\}$, the setup phase would need to occur each time with this static protocol. The results show that not only is there a lot of data being transferred between the two servers but that the timing values are quite high to perform an addition operation, which is less complex than performing a verification.

3.2.5 Summary

For the voting use case, partially homomorphic encryption is the best option, because it gives the greatest overall performance, especially in the cloud. Even when compared to plain text processing, partially homomorphic encryption can tally votes faster than decrypting and encrypting values. The zero-knowledge proofs are the only limiting factor in terms of performance but are necessary for the verifiability required. However, performance in the cloud is not the only factor, as client-side performance can be important. Because the fully homomorphic encryption scheme only encrypts a single bit, it would have good client-side performance, but the cost of the cloud computation is currently too high. The use of multi-party computation protocols show promise for other applications; however, their lack of flexibility and reusability make the homomorphic encryption schemes better suited for this type of application.

3.3 Extending to Survey System

A survey, for the most part, is a collection of ballots, one for each question. Therefore, the practicality of partially homomorphic encryption was tested for a survey system, raising some additional challenges over a simple ballot in terms of functionality. The codename for this application was Mau Surveys, where Mau can be translated to secure in Māori. Note that larger versions of screenshots given in this section can be found in Appendix C, and the source code can be obtained from Appendix D.

3.3.1 Creating a Survey

Disregarding the interface to customise the survey, the main challenge for the user-centricity of a system is the public and private keys. Many users of a survey system would have little knowledge of the concepts and importance of these keys. Therefore, asking a user to provide a public key for the survey could be too advanced for many users. Instead, the keys can be generated within the web application on the client side, such that the private key is not outside the clients' environment. Note that the management of the private key file is currently the user's responsibility, but could be stored in the cloud, using the user's password to encrypt it. This could even be achieved without the user's knowledge of the keys, hiding the idea of encryption altogether.

Figure 3.3 gives an example where the user has generated a private and public key. The private key is written to a file to be downloaded but is generated client side. This is implemented using a JavaScript library [152], meaning performance is slow due to how large numbers are handled. For example, creating a 2048-bit key pair can take over 4.5 seconds. The user could begin creating the questions and answers during this period, as can be seen in Figure 3.4. Similarly, the key pair could be generated as the page loads, meaning by the time the user has filled in the other details, a key pair is ready.

3.3.2 Completing a Survey

Given the slow performance of JavaScript for large number computation, it is currently not feasible to encrypt answers and generate proofs. Therefore,

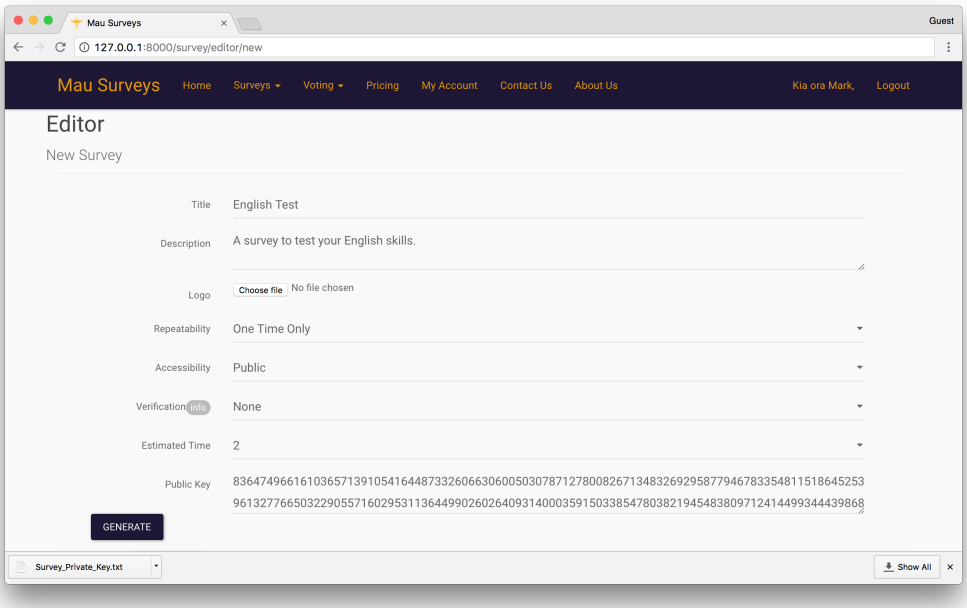

Figure 3.3: Generating the public and private key for a new survey

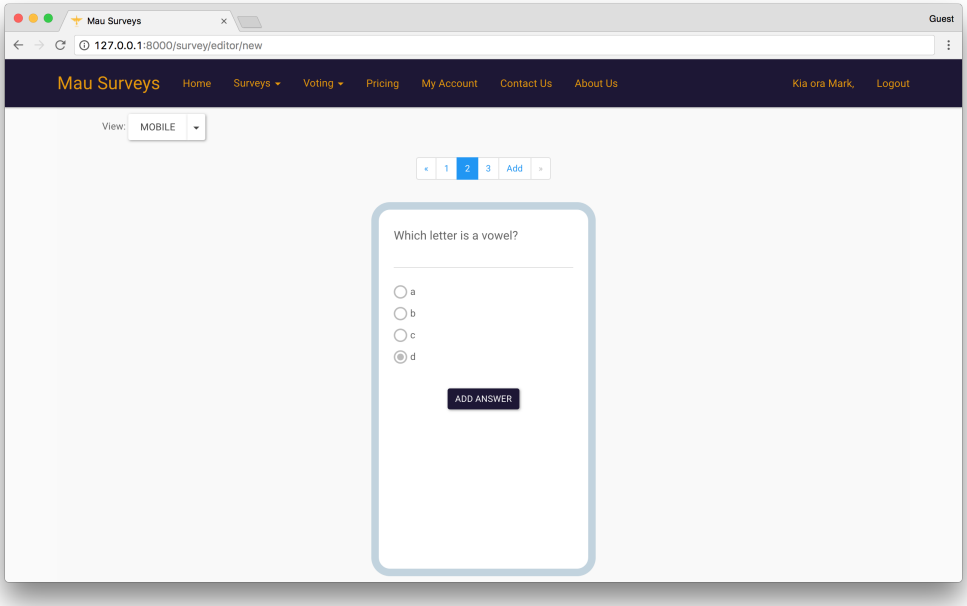

Figure 3.4: Interface for creating a survey

the portable native client was used for testing in Google Chrome, which allows native code to be executed and accessed by JavaScript (the Web Cryptography API does not support the operations required [153]). Note that a dedicated mobile application would not have this problem, but as Figure 3.9 shows, performance is still an issue. JavaScript can then be used to pass values to be encrypted and get the cipher and proof value back. This allowed a single answer to be encrypted in just over 500ms including passing to the portable native client and back to JavaScript. The performance of just the native code was a few magnitudes slower than in Section 3.2.2.

With the average reading time of under 300 words per minute [154], a question made up of 20 words will take over 4 seconds to read. Therefore, around 8 answers from the previous question can be encrypted while the user is reading the next question. For example, Figure 3.5 shows the computation occurring after the user moved to the next question. Note that the timing values are using the console timestamp, showing the timing difference between when the answer value was posted to the portable native client and parsed back by JavaScript. This hides the latency and computation from the user; only once the user completes the last question will they notice the encryption occurring.

Another factor to consider is the waiting time for a user to experience while their cipher answers are being verified and tallied. This will also affect the tallying server, which should not be vulnerable to many users submitting their answers at once. The solution implemented was for the server to store the answers into a queue, to be verified and tallied when possible. The user would later receive a notification that their answers had been added to the tally. This stops the user needing to wait for the tallying to occur. If the user is not behaving maliciously, their survey answers will have a valid proof. Therefore, it would be rare for a valid survey response to fail.

The server can then have workers running which read from the queue and process each survey response. First, a worker verifies all answers by checking the proofs. Then it proceeds to update the tallies, where atomic transactions occur when updating each answer's tally. However, note that multiple workers processing responses to the same survey may not give a performance gain. The conclusion here is that there are many factors impacting the overall performance and usability of a privacy-preserving cloud application, aside from

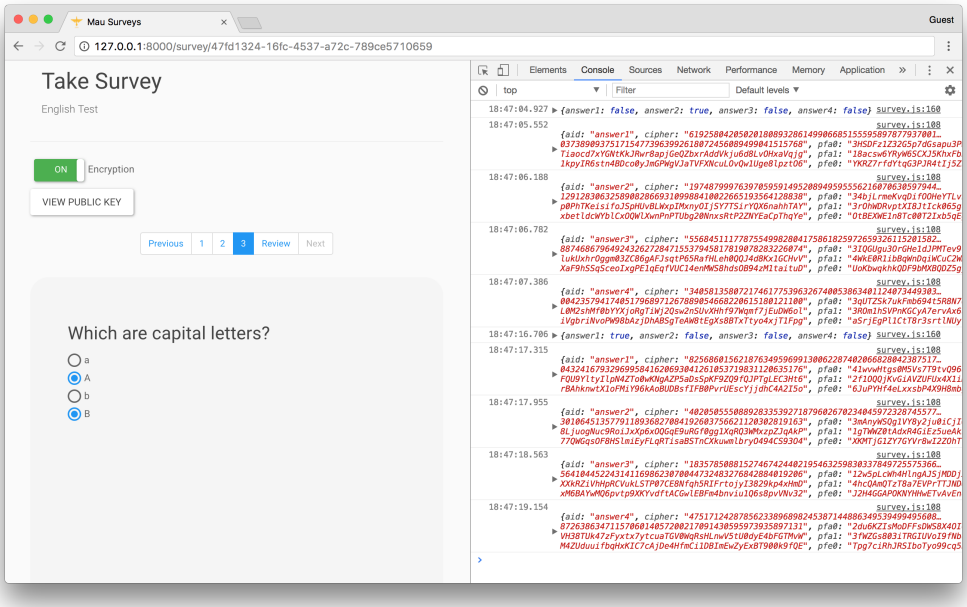

Figure 3.5: Example of a user taking a survey, with the encryption occurring in the background

just verifying and tallying values.

3.3.3 Reviewing a Survey

To review a survey, the user gets presented with a similar view as taking the survey. Only the owner of the survey can view this page. If the owner enters the decryption key, the results of the survey are shown, as seen for example in Figure 3.6. Decryption occurs client-side, with the cipher values already downloaded, as shown in Figure 3.7. Note there is an insecure vote option, for when a user chose not to encrypt their answers (Figure 3.5 shows the option to disable encryption). This was implemented to provide a balance between performance and security, giving the user a choice. The portable native client was used to decrypt the answers, where Figure 3.8 shows that it took 70ms to decrypt each answer. Therefore, 14 answers can be decrypted every second, which is acceptable.

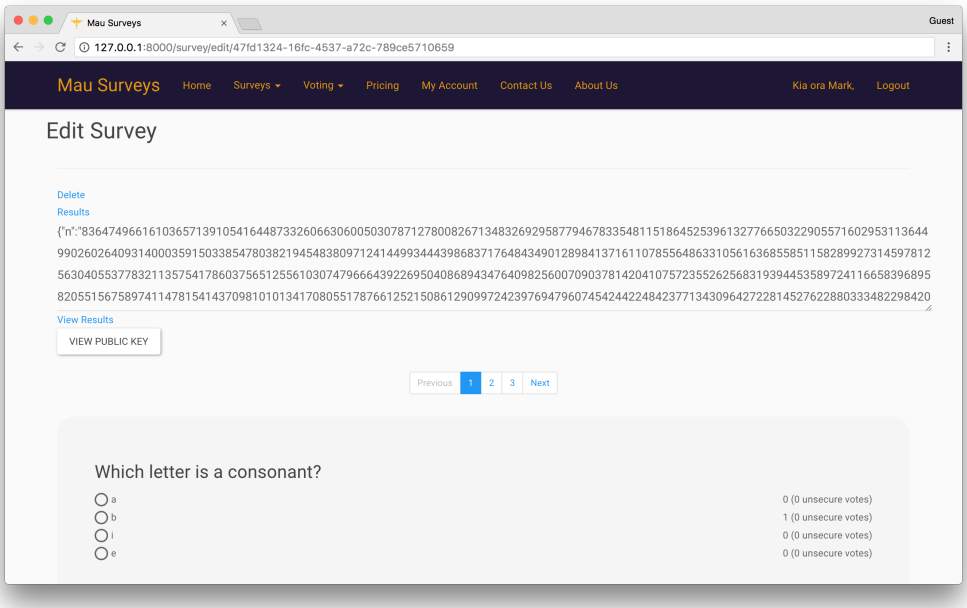

Figure 3.6: Example of a user reviewing the results of their survey

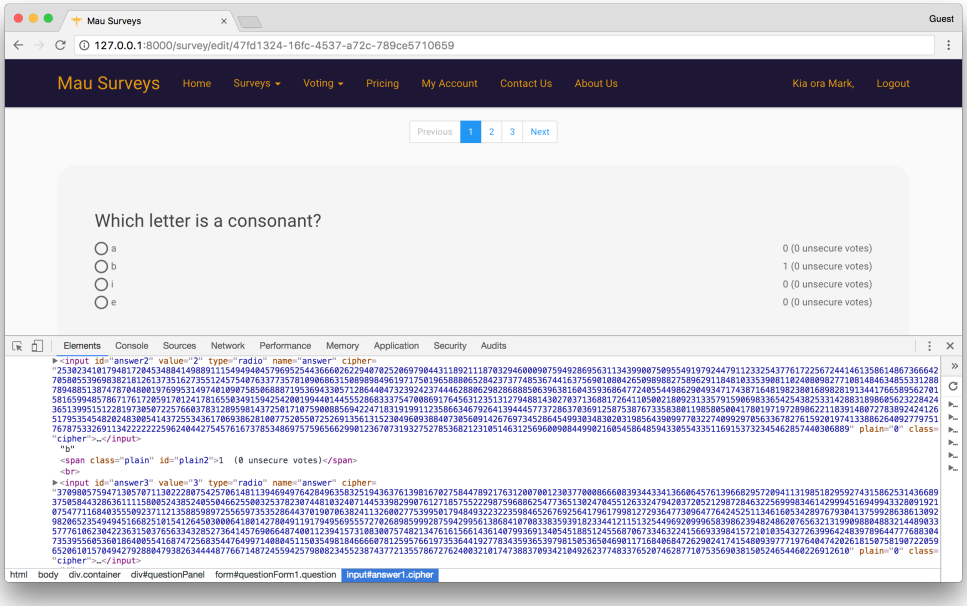

Figure 3.7: Example of how the cipher and plain-text results for a survey are presented to the user

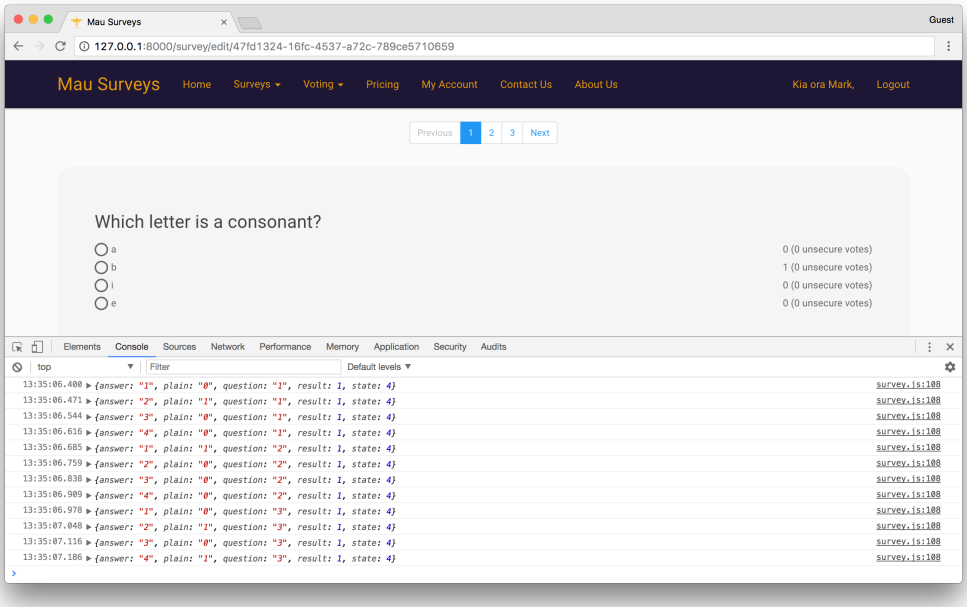

Figure 3.8: Example of the performance for decrypting each answer of a survey

3.3.4 One Answer Only

Multiple choice answers are exactly like having a voting ballot for each answer, where the answer is given a vote of yes or no. However, for questions where only one answer is allowed, this is more complicated, as there needs to be a proof of only one answer. This could be used in an election, where multiple candidates for are given.

For each answer, a cipher vote and proof are generated. Therefore, if the cipher votes are homomorphically added together, a new cipher vote is given, which should also $\in \{0, 1\}$, as shown in the equation below.

$$c_{sum} = \prod_{x=0}^{n-1} E(A_x)$$

This can be sent to the cloud along with the answer cipher values, where it can verify the summed cipher value. However, this does not guarantee that $c_{sum} \in \{0, 1\}$ because all the answers could be 1. To generate a proof for the cipher value, we need the random number used for encrypting the value. When encrypting each answer to the question, we can combine each random value

and use it to generate the proof for c_{sum}, as shown below.

$$g^{m_0} \times g^{m_1} \times r_0^n \times r_1^n \mod n^2 = g^{m_0+m_1} \times (r_0 \times r_1)^n \mod n^2$$

For example, below we show 4 cipher votes and the random values used, along with a small n value.

$$\begin{aligned} n &= 3331843951 \\ r_0 &= 3255715651 \\ r_1 &= 2642234083 \\ r_2 &= 3723126367 \\ r_3 &= 4266383843 \\ c_0 &= 620686413408095270 \\ c_1 &= 624310657855553550 \\ c_2 &= 9987196793370673622 \\ c_3 &= 2908154565487608414 \end{aligned}$$

The r values can be multiplied together, to give the r used for the sum of the votes, as shown below.

$$\begin{aligned} r_{sum} &= r_0 \times r_1 \times r_2 \times r_3 \mod n^2 \\ &= 41358922896747 54894 \\ c_{sum} &= c_0 \times c_1 \times c_2 \times c_3 \mod n^2 \\ &= 10286136893535093293 \\ &\equiv (n+1)^1 \times r_{sum}^n \mod n^2 \end{aligned}$$

The limitation with this approach is that it can only guarantee that zero or one answer was selected for that question. If multiple answers are required, for example 2, then this solution will not currently work. Instead, the answers in the backend could be possible combinations of answers. For example, a question with 10 answers where 2 can be selected gives 90 answers. This is computationally intensive for both the client and cloud. If the number of possible pairings is too large, two sets of answers could be encrypted, such

that each set can either have zero or one answer selected. This does then allow the same answer to be selected twice. Interface design can prevent this for the average user, but it cannot be guaranteed for a malicious user.

3.3.5 Linking Questions

Surveys can be used to provide statistics and analysis on topics, therefore the option to link answers together is critical. For example, does a person's favourite car brand relate to their favourite colour? Linking answers is also important for providing more advanced voting systems. There are two approaches for trying to support this feature: client intensive or server and decryption intensive.

Using an example of 2 questions, each with 4 answers, gives 16 possible combinations. This can then be treated in the backend as one question. Linking another question would give 64 options for the client to encrypt and generate a proof for each one. The performance on the client is the limitation for this approach, as well as the amount of data being sent from the client's device. Performance is reduced because only after all the questions being linked have been answered can the encryption occur, whereas encrypting a single question can occur while the user is answering another. When the survey loads, the client application or browser can start automatically generating all the cipher values and proofs, treating them all as zero. Therefore, only the linked answers need to be regenerated when the user has answered all linked questions.

Moving the computation and overheads to the cloud is another approach. The client encrypts each question as normal, along with the proof, and sends them to the cloud, where the cloud can perform the normal tallying. In addition, it can then link questions together. The concept is padding the answers together using bit shifting, as shown below.

$$\begin{aligned} &= (g^m \times r^n)^2 \mod n^2 \\ &= g^{2m} \times r^{2n} \mod n^2 \\ &\equiv m << 1 \end{aligned}$$

For example, combining two questions, the answers from question 1 and from question 2 $\in \{0, 1, 2\}$ (bit length of 2) can be summed. Therefore, with a

64-bit integer, 32 results can be padded into a cipher value, meaning a new cipher value is needed for every 32 participants. With a million participants, using a 2048-bit key, a few megabytes are needed for each combination. The challenge is that for the proof, the actual value encrypted in Section 3.2.2 $\in \{1, 97\}$, meaning the sum $\in \{2, 98, 194\}$ for linking two questions. These are $2 = 0000\ 0010_2$, $98 = 0110\ 0010_2$, $194 = 1100\ 0010_2$ where the lower 4-bits are the same. Therefore, when bit shifting, instead of shifting by 8, we can shift by 4. Apart from the first 8 bits, the most significant bits are $2 \equiv 0010_2$, $98 \equiv 1000_2$, $194 \equiv 1110_2$. Therefore, a 64-bit integer can store 15 participants.

To get the linking results, the survey creator decrypts all packed values and can tally the linked results. This is another disadvantage of using the padding approach. Time and cloud storage can be saved by not recording linked values for the last answer to the question, because these can be solved using the other values and final tallies of each answer. The two approaches are contrasting, because practicality is either achieved for the client or the cloud and survey creator.

3.3.6 Other Trust and Privacy Issues

In creating the survey cloud application, there were a number of issues that still arose even with the data encrypted through to the survey creator. The cloud provider and survey company cannot see any data, but still can learn information, where the survey title, questions, and answers can leak information about the user taking the survey. One solution to protect the entire survey from the cloud provider is to have a password or some form of key obfuscating the survey. Users could also be given unique codes to unlock the survey, such that they do not need to create an account.

The major trust issue discovered is guaranteeing the public key used to encrypt the survey on the client device is the same generated by the survey creator. The balance between making the application user-centric and secure came into question. The solution implemented was to give the user an option to view a hash or the entire public key when taking the survey. This would prevent the public key being changed in the database. The survey creator would then need to make the key and hash publicly available, for example on social media, but this challenge highlighted how difficult it is to hide the

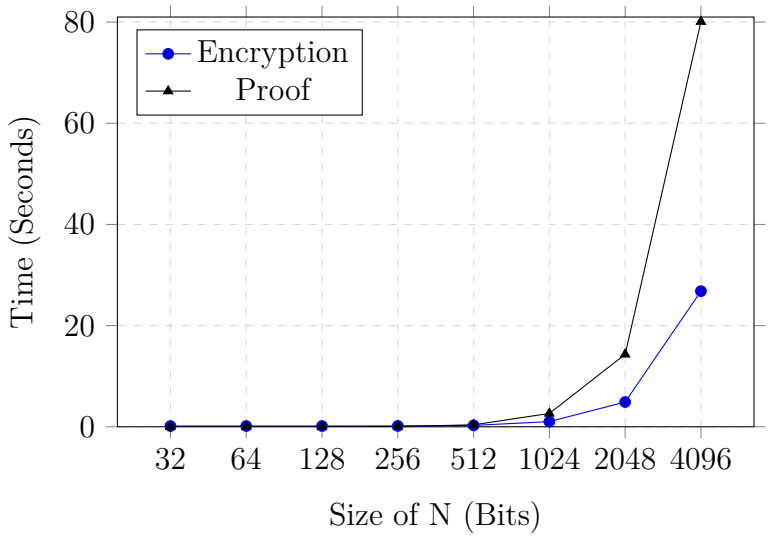

Figure 3.9: Mobile voting performance on an iPhone 5

encryption from the user.

3.3.7 Real versus Virtual World Practicality

Presented results for partially homomorphic encryption thus far suggest good performance. However, when tested on an average mobile device, the computational intensity of computing large numbers is highlighted. Figure 3.9 shows that generating a proof can nearly 20s for a 2048-bit key. In terms of computer science, seconds to minutes for obfuscating a vote on the mobile device is seemly very slow. However, with the use case of voting, the algorithm and system performance are not the only components to consider for practicality. The following is an example of the typical procedure a voter would have to undertake: commuting to the voting station, queuing at a voting station, authenticating themselves, and filling in voting papers. Therefore, if the encryption and proof generation takes a few minutes on a client's mobile device, it is still time saved overall. A user could be entertained with a game or could be notified once the computation is complete. The convenience and time saved validate the minutes of computation time as being practical.

In contrast, with a survey, practicality is defined differently because completing a digital survey needs to be as fast as possible, otherwise participants are less likely to finish it. For example, with non-secure survey systems, the

user experiences near-zero latency when completing the survey. As discussed in Section 3.3.2, a lot of the computation can be hidden as the user completes other questions. However, with partially homomorphic encryption, depending on the device and key size, the user could experience a few seconds or minutes before being able to submit the survey. Again, the survey could be submitted automatically once encryption is complete. However, the same technique is not as practical for a survey system, as it is for a simple ballot, where the cloud performance is practical, but the client side is still problematic.

3.4 Summary

This chapter has shown that the most practical privacy-preserving solution is currently partially homomorphic encryption, with multi-party computation practical for some types of applications. The use case of secure voting and the extension into a secure survey system has shown some aspects for what it means to be practical and some limitations of existing solutions. Different schemes will be better suited for some applications than others. Therefore, practicality depends on the application; however, focusing on cloud performance, partially homomorphic encryption takes under one millisecond to compute a homomorphic operation with a key size of 2048-bits. Combined with the proof, this gives around 35ms for a vote to be added to the tally. This should be the target of any scheme claiming to be practical in the cloud. However, Section 3.3.7 raised the issue of client performance, and fast encryption times are also required, otherwise even with practical cloud performance the application is still not usable.

4

Privacy-Preserving Encoding

In researching Hypotheses 1 and 2, there was no direct outcome to answering the research question for this thesis. However, the simple fully homomorphic encryption algorithm explored in Appendix B did offer search capabilities, spawning the idea of using encoding for privacy-preserving processing. Instead of encrypting characters into cipher values, encoding them into groups or "bins" could allow search capabilities while protecting privacy. This chapter explores that idea, resulting in a tentative conclusion for Hypothesis 3.

4.1 Bin Encoding

An approximate string searching scheme "Bin Encoding" is presented as an initial comparison between encoding and encryption [22]. This is a lossy encoding scheme—a simple trapdoor—that maps characters individually to bins (an extension of the simple substitution cipher used by Mary, Queen of Scots). There are several bins, and multiple characters map to the same one. Hence, the original string cannot be easily obtained from its encoding. For example, below is a mapping with three bins A, B and C:

$$\{a, b, c, d, e, f, g, h, i\} \Rightarrow A$$
$$\{j, k, l, m, n, o, p, q, r\} \Rightarrow B$$
$$\{s, t, u, v, w, x, y, z\} \Rightarrow C$$

(This example is for illustration; in practice, we envisage many more bins

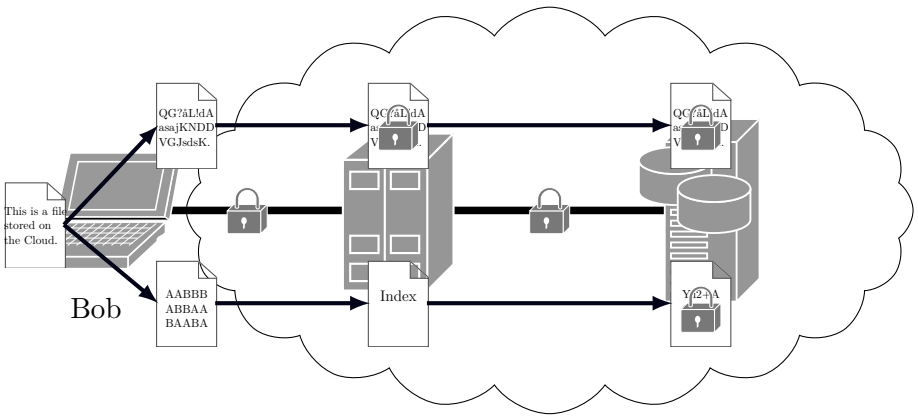

Figure 4.1: Personal user system model for Bin Encoding

to reduce the number of false positives when searching.) Relative to this mapping, the encoded values for *hello* and *world* are *AABBB* and *CBBBA* respectively, which can be obtained using Algorithm 1. Apart from *world*, another possibility for *CBBBA* is *snore* (amongst others). However, these possibilities can only be generated by someone who knows the bin mapping. Given the encoded value but not the mapping, there are countless possibilities for *CBBBA*, such as *hello* (even though the above bins map it to *AABBB*). The user's data is protected by hiding it in many possible bin combinations ($> 10^{20}$).

Algorithm 1 Bin Encoding

```
1: function BINENCODE(string, binmap)
2:     estring ← ''
3:     for i ← 0 to len(string) do
4:         c ← lowercase(string_i)
5:         if c in binmap then
6:             estring ← estring + binmap_c
7:     return estring
```

4.1.1 System Model

The scheme is aimed at the personal cloud end user, with Figure 4.1 showing a typical use case. Bob is saving a file in the cloud and would like it to be encrypted, while retaining the ability to search its contents at a later date. Before transmitting the file, his device encodes it; the computational expense

is trivial. Bob separately encrypts the file using a secret key, and sends both encrypted and encoded versions to the cloud service. Filenames are encrypted, while smaller documents can be padded.

The service stores Bob's encrypted document and adds the encoded version to its index, which is encrypted when not in use. It manages the encryption key for the index, but only Bob knows the key for his actual documents. When searching, he sends encoded queries to the cloud service, which uses the index to determine which documents satisfy the query. Bob's data is protected, because the cloud service cannot decipher either the encoded documents or the encoded search terms.

4.1.2 Removing Special Characters

To harden Bin Encoding against language attacks, special characters, such as spaces must be removed, because they reveal too much information. For example, if spaces are visible, encoded words of length 1 will correspond with high probability to either i or a (in the English language). If the space character is encoded, its bin will occur unusually often. For example, a high-frequency letter like T occurs in English around 9% of the time [155], whereas space has a far greater frequency of $16 - 30\%$, assuming an average word length of 5. This may make it easy to detect which bin contains the space character, leading again to the problem of single-letter words. The index is therefore based on bin positions in the document and not words like a traditional index.

4.1.3 Approximate String Searching

By encoding characters individually, approximate string searching can be supported. Using the above mapping, *hello* encodes to *AABBB*. But *hallo* also encodes to *AABBB*, so they would still match. Furthermore, if h and e were swapped, the encoded result would remain the same. Of course, this is an unrealistically simplistic example, and with more bins approximate searching will need to take into account how many bins are different, or transposed.

Modern operating systems support built-in spell checking, which perhaps reduces the importance of approximate string searching. However, this adds an overhead to the client, it has a limited dictionary, and users are taxed by

having to confirm corrections. Thus, it is advantageous for searching schemes to support approximate string matching.

4.1.4 False Positives

Searching for a string over unencrypted data rarely returns false positives. However, when both query and data are encrypted, a cloud service cannot determine how accurate its results are. Lossy techniques such as Bin Encoding make it even harder to prevent false positives, for not only does the service have no knowledge of the data and query, but it cannot be sure that two identical encoded values are actually the same. Furthermore, an approximate search increases the potential for false positives if an exact match is not found. However, by looking for patterns, Bin Encoding still produces accurate results, as will be shown in Section 4.3.

4.1.5 Building the Index

The scheme allows the index to be built by the cloud service, rather than the client. The client only needs to encode the documents, which adds negligible overhead, and encrypt them. This allows multiple devices to update documents at the same time. In contrast with trapdoor searching, delegating index operations to the cloud service allows it to implement more advanced search facilities and obtain more accurate document rankings, because the locations of query term hits are known.

Just how the index is built is left open, and the details are beyond scope here. However, given the cloud has no knowledge of spaces, we recommend an n-gram approach, where n would be the number of bins. For testing, we built the index using grams of the same size as the query, although in practice it is likely that fixed sized grams such as bigrams or trigrams will be used. Note that overall performance will depend on how the index is constructed and the number of documents in the index.

4.1.6 Generating the Bin Mapping

There are many ways of generating the mapping of characters to bins. A simple solution is to generate it randomly and save it for future use. However, if a user

accesses the cloud from a different device, it will also need the map—however, it is undesirable from a user and security point of view to transfer the map from device to device. Instead, we recommend creating the mapping from a password.

Algorithm 2 Bin generation example

```
1: function BINGEN(password, salt, bins)
2:     binmap ← {}
3:     hex ← Hash(password + salt)
4:     i ← len(hex) − 1
5:     for c ← A to Z do
6:         while c not in binmap do
7:             p ← int(hex_{i to i−1}) mod bins
8:             i ← i − 2
9:             if len(binmap_p) <= ⌈26/bins⌉ then
10:                binmap_p ← binmap_p + c
11:    return binmap
```

This can be done in many ways; as a concrete example we describe a simple method, shown in Algorithm 2, that uses a hash function. The password, along with some user-specific salt, is hashed into a large number. This is expressed in hexadecimal and treated as a string, parts of which are converted to an integer in modulo N, where N is the number of bins. This is used to assign the letter to a bin. If equal bin sizes are required and the bin is full, the process is repeated. If the entire hashed value is consumed, we could return to the beginning and re-hash with an offset (not shown in Algorithm 2).

4.2 Privacy with Bin Encoding

Given Bin Encoding is a simple lossy encoding technique, it cannot be classed as encryption. However, encoding can also protect privacy and secure data. The protection is analysed in Sections 4.2.1 and 4.2.2 for frequency and brute force attacks respectively.

4.2.1 Frequency Attack

The simple substitution cipher used by Mary, Queen of Scots, was broken via frequency analysis, which led to her execution in 1587 [156]; this is why

Table 4.1: English letter frequencies

A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**
7.61	1.54	3.11	3.95	12.62	2.34	1.95	5.51	7.34	0.15	0.65	4.11	2.54
N	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
7.11	7.65	2.03	0.10	6.15	6.50	9.33	2.72	0.99	1.89	0.19	1.72	0.09

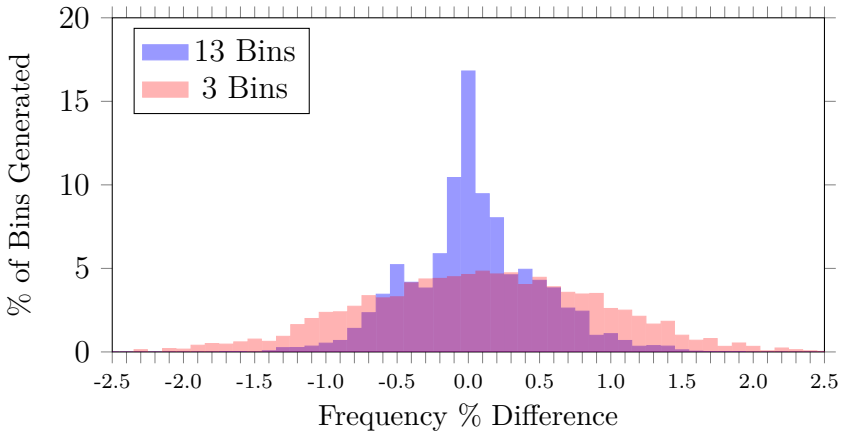

Figure 4.2: Difference in calculated and actual bin frequencies

frequency analysis is covered in this section. Analysing the frequency of a bin occurring gives an estimation of the letters mapped to it. For example, given the letter frequencies in Table 4.1 [155], if a bin occurs at a relatively small frequency, it is more likely to contain letters that also have a smaller frequency. This also gives a reduction in bin combinations for a malicious user to try, because certain combinations do not fall within the estimated frequencies calculated.

Figure 4.2 shows the difference between the estimated frequency obtained from counting bins in the index and the actual frequency of the letters in each bin. These results show that with enough encoded documents indexed, it is possible to predict within $\pm 2.5\%$ of the actual letter frequency. They also show that a smaller number of bins for the index is harder to estimate.

Figure 4.3 shows the distribution of 100 million unique random bins for a 3 bin configuration, giving a total of 300 million bins, each containing $8-9$ letters. The average summed frequency for a bin is around 33.3%, even though the English letter frequencies vary. Therefore, when generating the bin mapping, we can check if the bin frequencies are within the majority of all possible bins combinations. For example, using 3 bins, the scheme might only

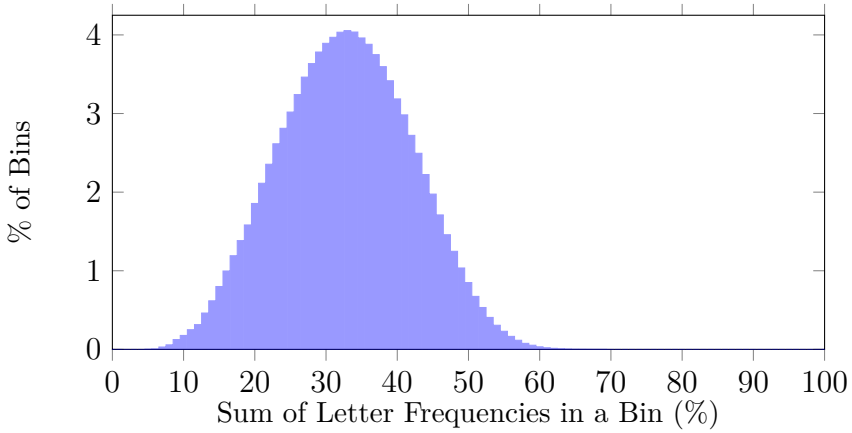

Figure 4.3: Letter frequencies in 300 million randomly generated bins

accept bins with frequencies between 20% and 46%. This means that even if a malicious user knows the frequency of each bin to ±2.5%, it still gives over 20% of all possible bin combinations. Note this experiment only contained bins of even size (8-9 letters), where an implementation having a variation in the number of letters per bin would have more possible bin combinations.

4.2.2 Brute Force

There exists a finite number of states for a mapping; therefore, a malicious entity need only find which. With an even number of characters per bin, the expression below determines the number of possible states where N is the number of characters, b is the number of bins, and $N \mod b = 0$. For example, with the set $\{A-Z, 0-9\}$, gives $N = 36$. When $b = 3$, it results in $\approx 3.38e^{15}$, and if $b = 12$ then there are $\approx 1.71e^{32}$ possible states. An uneven number of characters per bin grows these values further.

$$\prod_{i=1}^{b} \binom{N-(N/b)(i-1)}{N/b}$$

With the correct state or mapping, because Bin Encoding is lossy, there are still many combinations that the original plain-text value could be. With $b = 3$ and $N = 36$, then there are 12 possibilities for each bin. With a query of length 5, there would be 12^5 possibilities, before considering if it is a valid word

or phrase. An encoded document with only 100 characters would therefore have $\approx 8.28e^{107}$ possibilities. Note that with 100 characters, and given spaces have been removed, word lengths are not known. If the mapping is not known, trying to solve an encoded document is hard ($\approx 8.28e^{107} \times 3.38e^{15} = 2.80e^{123}$). The challenge is knowing the solution is correct. For example, using the prime number theorem to estimate the number of prime numbers that are 1024-bits gives many more possibilities $\frac{2^{1024}}{\log(2^{1024})} - \frac{2^{1023}}{\log(2^{1023})} \approx 1.27e^{305}$. However, the difference for encrypted text is it is usually obvious when the decryption key is found, where with Bin Encoding there are many possible solutions even with knowledge of the mapping.

An example can be seen in Table 4.2, where all 5 letter words (6919) from a dictionary [157] were tested against each other for collisions using the same mapping. This shows that with only a few bins, there is the potential for some false positives in results. However, it also means that for trying to recover the plain text, that there are a number of options even if the mapping and word length are known. Results from another experiment in Tables 4.3 and 4.4 show the large number of potential matches if the mapping is not known. Each word was encoded into x bins using ASCII values modulo x, and tested for a match against the query word, giving the same mapping column. The other mapping column was computed by encoding the query and testing if each 5 or 10 letter word could be a potential encoding. The criteria for a potential match was that no character gets encoded to different bins. For example, *hello* is not a potential match for the encoded query *ABBAC* because *l* is split across two bins. Table 4.4 also includes two 5 letter words which still have potential matches with 10 letter words.

4.3 Bin Encoding Results

The previous section detailed how difficult it is to reserve the mapping process for Bin Encoding. The functionality of a search is still present though, as will be shown in this section. String searching over a document is given in 4.3.1 before language analysis in the form of collisions is given. The lower the collision rates for a language, the better results Bin Encoding will return.

Table 4.2: Collisions between five letter words

Number of Bins	Average Collisions	Percentage
1	6919	100%
2	238.586	3.448%
3	28.473	0.412%
4	9.759	0.141%
5	3.810	0.055%
6	2.824	0.041%
7	2.092	0.030%
8	1.684	0.024%
9	1.332	0.019%
10	1.501	0.022%
11	1.195	0.017%
12	1.189	0.017%
13	1.128	0.016%
14	1.256	0.018%
15	1.143	0.017%
16	1.137	0.016%
17	1.084	0.016%
18	1.055	0.015%
19	1.029	0.015%
20	1.085	0.016%
21	1.021	0.015%
22	1.009	0.015%
23	1.007	0.015%
24	1.009	0.015%
25	1.000	0.014%
26	1.000	0.014%

Table 4.3: Five letter word queries

Word	Bins	Same Mapping	Other Mappings	% in Dictionary
hello	3	16	5374	77.94%
hello	13	1	5058	73.15%
world	3	80	5407	79.34%
world	13	1	4598	66.50%
paper	3	20	4963	72.05%
paper	13	3	4871	70.47%
zoned	3	106	5348	78.86%
zoned	13	1	4598	66.50%

Table 4.4: Ten letter word queries

Word	Bins	Same Mapping	Other Mappings	% in Dictionary
blacksmith	3	1	1994	16.79%
blacksmith	13	1	635	5.35%
underwater	3	2	2861	24.10%
underwater	13	1	1272	10.71%
helloworld	3	0	2629	22.13%
helloworld	13	0	1212	10.20%
purringcat	3	0	2156	18.15%
purringcat	13	0	893	7.52%

Table 4.5: Exact matching in a document

Search Query	First Result	Second Result	Third Result
internet	internet (100%)	interest (42%)	intercon (33%)
ethics	ethics (100%)	ewhich (45%)	ethica (45%)
privacy	privacy (100%)	videdby (28%)	nyactiv (28%)
hackers	rkthere (28%)	isthehe (28%)	hersche (28%)
underwater	waterreser (33%)	underscore (26%)	-
sldnfs	-	-	-

Table 4.6: Approximate matching in a document

Search Query	First Result	Second Result	Third Result
internat	internet (53%)	informat (53%)	interest (42%)
intranet	internet (42%)	infrastr (33%)	intended (33%)
intrenet	internet (66%)	interest (53%)	inginter (53%)
ethiks	ethics (45%)	fthiss (45%)	ethica (33%)
ehtics	ethics (33%)	effect (33%)	-
priatcy	privacy (38%)	leinthe (38%)	beingof (28%)
private	privacy (38%)	provide (38%)	privile (38%)

4.3.1 Searching over a Document

The document used to obtain these results was "Ethics and the Internet RFC1087" [158]. For a single index, 13 bins were used, with all bin mappings randomly generated at runtime. The primary ranking used was the ngram search function from the Python ngram library [159]. Results are the size of the query, so some results are segments of words, or even two parts of a word, because spaces are not known.

Table 4.5 contains six queries; the first three queries exist in the document

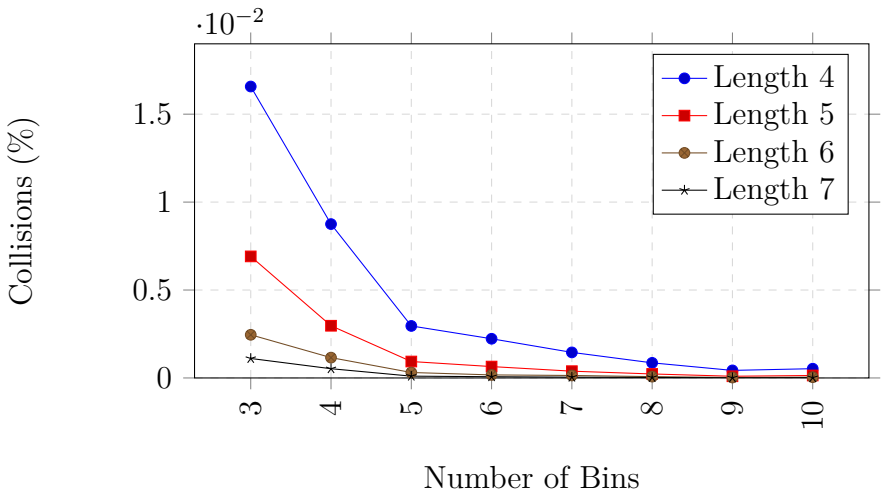

Figure 4.4: Average word collisions rates for English using modulo bin allocation

and this is shown by the results giving 100% matches. The remaining results (second and third) can vary depending on the bin mappings; however, if the query is not misspelled and the search term is in the document, a 100% match always occurred in our testing. The bottom three queries are terms which are not in the document and hence do not get strong matches.

The results for approximate searching are given in Table 4.6 where a selection of incorrect queries is given. These results show that the cloud is still able to match a word/phrase that the user could be meaning. Note that because approximate matching is more challenging for Bin Encoding, even a single wrong character in the query can lead to varying results. However, the important aspect is that the correct result is returned to the user even if it is not the first match, where a quick filter on the client side could be used to move it up the rankings.

4.3.2 English Collision Rates

In order to test false positives, we computed all collisions (possible false positives) between words of length 4, 5, 6 and 7. Figure 4.4 shows the average collision percentage of English words from the dictionary [160].[1] Looking at the percentages, using 3 bins will clash with less than 0.017% of all words. Then, as the bin sizes increase, the rate of collisions drops to average just over

[1]Figures 4.4, 4.5 and 4.6 do not start at 0 because they either had too high collisions in Table 4.2, or for the case of 0 and 1, are not usable.

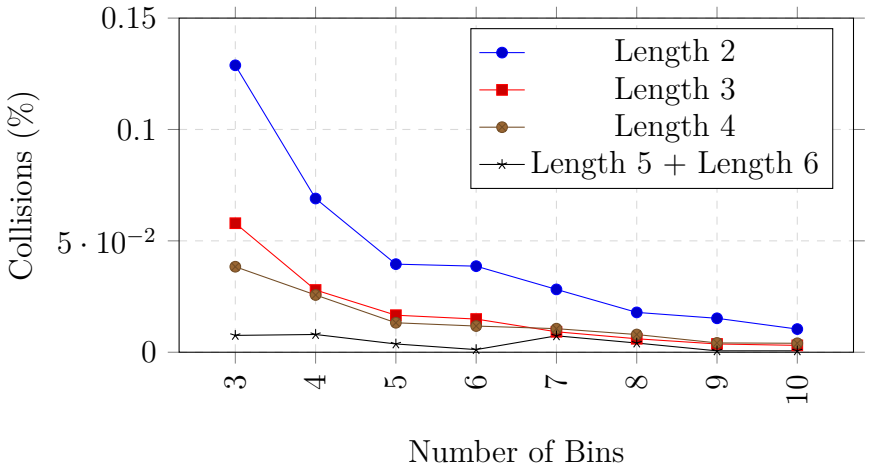

Figure 4.5: Word collisions rates for Pinyin using modulo bin allocation

a single word. This means that using a larger number of bins for the index gives few false positives.

4.3.3 Mandarin Collision Rates

Given the complexity of Mandarin and other Chinese dialects, even basic keyword searching can be challenging [161]. A single character represents a whole word or phrase, and can vary depending on the characters surrounding it, unlike English. Therefore, we used *Hanyu Pinyin* to romanise Chinese characters, by spelling out their sounds. This is a lossy approach, as some characters can produce the same Pinyin output; however, Bin Encoding itself is already lossy. Once the characters are converted, encoding and searching can proceed like other languages.

Pinyin produces relatively small words (sounds) from a single character, which effects the collision rates, as can be seen in Figure 4.5 (tested on a small dictionary [162]). However, because words are a combination of sounds, the collision rate will drop as they are combined, like the last entry in Figure 4.5. This aligns the collisions rates to other languages. Therefore, exact searching can be supported by Bin Encoding, but approximate searching will be more difficult. If the user input is Pinyin, then approximate searching will be able to support errors. However, if the user input is a Mandarin character, which is then automatically converted to Pinyin, approximate searching will not work.

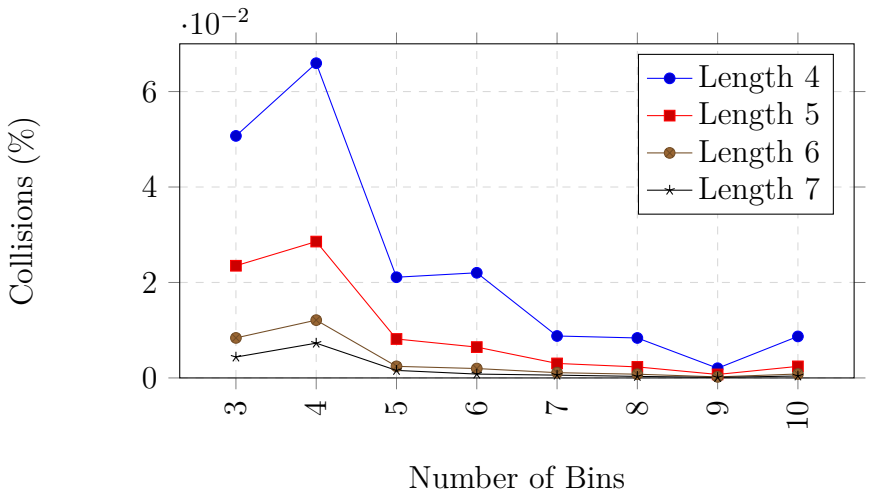

Figure 4.6: Word collisions rates for Māori using modulo bin allocation

This is because an error with the input could cause large variations to the Pinyin output.

4.3.4 Māori Collision Rates

Spoken by the indigenous people of New Zealand, Māori is a variation of historic eastern Polynesian languages. It has an interesting property, were each syllable ends in one of 5 vowels (10 if we include long vowels which are indicated by a horizontal bar). Therefore, given an encoded Māori query, the last bin must contain either a short or long vowel.[2] This opens up a new attack vector that would not be possible with the English language. However, given the large number of vowels and that the cloud does not require knowledge of which language is used, it is not a major issue.

Figure 4.6 shows collision rates between words of equal length where the bin mapping was performed using a modulo allocation. Each word length shows a spike when using 4 bins. This is because of the poor distribution of vowels across bins, as 4 out of the 5 short vowels give the same modulo result. Using a random bin allocation as in Figure 4.7 gives more desirable results and shows the importance of the bin mapping generation.

[2]The dictionary was obtained from Dr. Te Taka Keegan (tetaka@waikato.ac.nz).

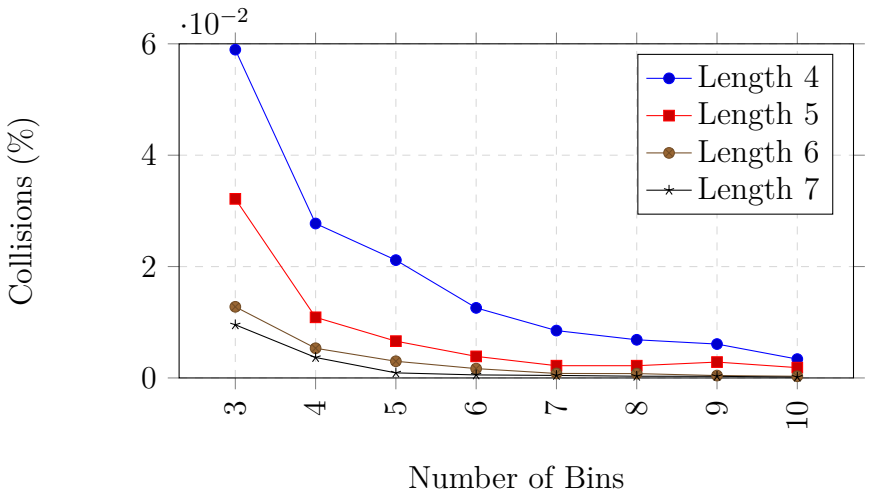

Figure 4.7: Word collisions rates for Māori using random bin allocation

4.4 Distributed Bin Encoding

As previously discussed in this chapter, a larger number of bins will return more accurate results from encoded queries, because more patterns are observed. However, having many bins (>13) opens both index and queries to a wider range of attacks. In the extreme case, 26 bins are simple to attack, because there is a one-to-one mapping between characters and bins (assuming the alphabet is $\{A-Z\}$), enabling standard frequency and language attacks—although removing spaces makes these somewhat more difficult. In practice, the index will be built using a smaller number of bins (<6) to harden it against such attacks.

4.4.1 System Model

The proposed distributed system model stores data in different "environments" that are isolated from one another in such a way that if one environment is compromised by a malicious user, they will not be able to use the information or privileges gained to compromise another environment (possibly hosted by different cloud service providers as well). Figure 4.8 shows a distributed index model containing four separate environments that isolate data. Three of these environments, I, II, and III, contain indexes to the same documents built using different bin mappings. By combining search results from each index, more accurate results can be returned. For example, if one index gives a match on

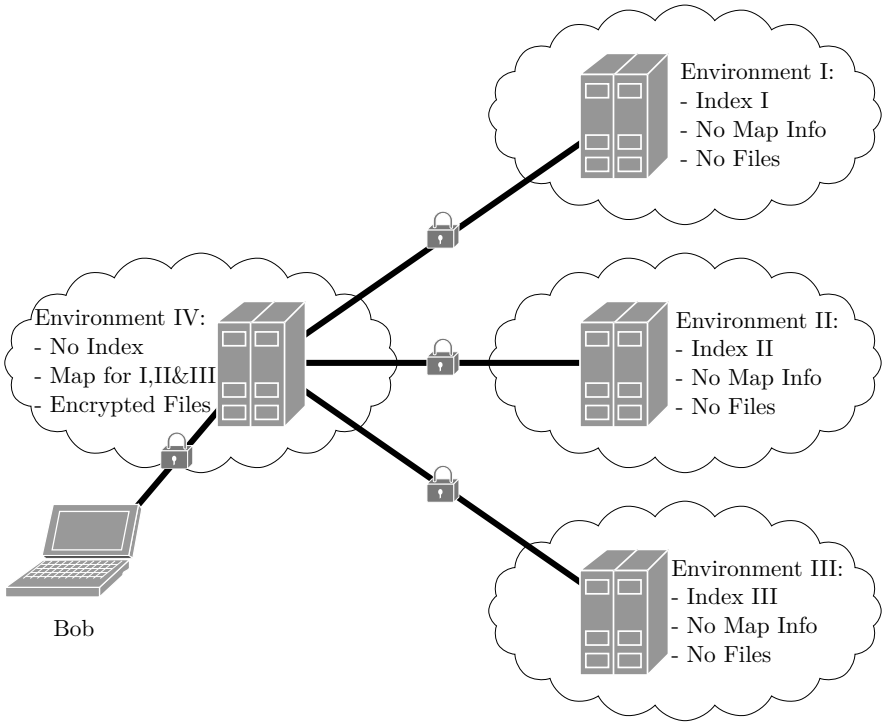

Figure 4.8: Distributed index system model for bin encoding

part of the document but another does not return the same match, this is likely to be a false positive.

Environment IV in Figure 4.8 is responsible for distributing the query to the indexes and for combining the results to return to the user. When the user sends a query or edits a document, the number of bins for encoding can be large (for example 13). When Environment IV receives encoded values, it uses a randomly generated mapping, different for each user, to further encode the already encoded values for each environment into a smaller number of bins. Although the user's mapping is never sent to the cloud, each index has a different mapping. Environment IV also stores the encrypted files received from the user (although they could be placed elsewhere). The distributed index provides better security than a single index, because a compromise on a server with an index only has a small number of bins making it harder to recover the plain-text values.

4.4.2 Results

Instead of using 13 bins for the index, the client encoded documents into 13 bins for sending to the distribution server, before being converted into the three indexes, where each used 3 bins with different mappings. Results are not shown here because they essentially replicate the ones achieved in Section 4.3.1. This proved that the distributed model is the superior model, as it provides more privacy and accuracy than a single index.

4.5 Limitations of Bin Encoding

There are limitations for using the lossy mapping function, some of which are mentioned in this section.

4.5.1 Results Known

Allowing the cloud to know if a result was found is an advantage and disadvantage. For large indexes, knowing if a document matches means less data is transferred back to the user. However, better privacy would be achieved by the cloud not knowing if a result is found. Given the file content and name are encrypted, along with the queries not known, the cloud knowing the number of matches is acceptable for this scheme given the focus is on practicality.

4.5.2 Randomness

A property of strong cryptosystems is their ability to provide randomness, such that the same plain-text value produces different cipher values. If the cloud does not know if a match is found, this is more important. Currently, Bin Encoding does not support randomness for queries; however, two identical encoded phrases can be different plain-text values, especially with a small number of bins such as with the distributed model. Randomness for Bin Encoding would involve randomly corrupting bits of the queries where multiple queries would be needed to filter results. However, this was not tested or explored.

4.5.3 Light Obfuscation

To make dictionary and frequency attacks more difficult, we propose the idea of lightly obfuscating each word before it is encoded. This must be done at a character level, to preserve approximate searching. Obfuscating a couple of characters per word results in characters being encoded into different bins. This heavily reduces the accuracy of frequency attacks. Dictionary attacks are also hindered as each possible light obfuscation method must be considered when testing each word in the dictionary.

Algorithm 3 shows an example where every second character is XORed with the first letter. The obfuscation frequency (every second character) and function (XOR) should vary for different users, like the bin mapping. The key can be an arbitrary value, but for this example we use the first character. This is because in the majority of Google's recorded misspellings of "Britney Spears" [135], the first character is correct.

Algorithm 3 Obfuscate word before encoding

```
1: function obfuscateWord(word, freq, func)
2:     key ← word_0
3:     eword ← ''
4:     for c in word do
5:         if index(c) mod freq == 0 then
6:             c = func(key, c)
7:         eword ← eword + c
8:     return eword
9:
10: binEncode(obfuscateWord(word, 2, XOR), binmap)
```

4.5.4 Padding

Small documents which are accessed often could reduce the attack space for a dictionary attack: for example, as in a shopping list as shown below:

ShoppingList.txt

milk

eggs

bread

red bull

A solution to hide these smaller documents is randomly padding the encoded output on both sides. It is important that the length is random, otherwise all the small documents will have the same padded length, making them easy to identify. When generating the padding bins, it is possible to create patterns which will be returned when searching. Preventing false positives for the file is simple, as we can check that the random bins do not have the same patterns as the original encoded string. For example, for the shopping list we checked three bin patterns, and the padded result can be seen below, where the original string is in blue.

```
GKMGHDHCCNGADANJACEHGIFGLFDBGCNCDJDIHEAIBNAALBF
MCIFDBKNHMDMLGGBJMJIAIEBAJLHNKCJECBMLMDFEFLDLJG
AHLHGJMIHIEEEDGFJAMBFKAHAHBNGCAHMEDCMAFFBDAADMK
ADBHAFHMILLJFDLBILDMDIKDCCBHBLJFBJCAJJJKIBEFIAK
FILHEEMCDCJKEGALLBCCBBGHFCACELGKJLKNKDHNKEHFJNB
INJNNKDMFLECEGGKEILDMKMAFHBCDBFNJHDLHDAIMMMIBKK
FIFJFMLJNFMJCFAACGLKIMKCHEEJDECGDEMMHNIFGIKIJCJ
NDEJICJIKCDKIICHAKEBBADKAKBHGCCHNFNFIDLLFDJJKEN
FCCLAHKKFKKLDLAAMFLMHNMIAIEJKMGILMGMABJCNFCIJFA
IAGKCMGJENCFEFENCEGGKGAHDDNIEKCNICJJAKICILBFJCH
            BCLLJDNMAHNKINFEEMDJCNH
```

Padding the shopping list to around 500 characters 1,000 times (using different randomly mappings), 4 and 5 letter dictionary words were encoded and compared for exact matches. On average, 53×4 letter words and 95×5 letter words were falsely added. This is only 0.0153% and 0.0131% of the possible words respectively. These results are within the same percentages of collision rates, making padding a feasible solution for small documents.

4.5.5 Preview Results

Providing the user with preview results on what was matched – like the Google search engine – is very difficult for encrypted data. If a document is encrypted using a technique where the entire document must be decrypted at once to get the plain-text, then there is a lot of overhead (downloading the file and decrypting it) to get a snippet of the document. To improve this overhead, the

document could be encrypted in smaller chunks so that only a quick decryption is required. However, this then adds overhead to the encryption phase.

Stream ciphers [163][164][165][166] offer greater performance than traditional schemes but can sacrifice the level of security. However, they offer a useful property for encrypted search. If no block chaining is used, such as where each character is XORed with the output of a pseudo-random number generator, then it is possible to only download and decrypt the part of the document where a match was found. This technique decreases both the encryption and decryption steps, while not requiring the entire file to be downloaded.

The current limitation is that Bin Encoding returns an index into the file where the match was found without special characters, meaning the index does not factor in spaces. Therefore, a third file must be uploaded, where special characters are removed, and it is encrypted using a stream cipher. This allows for correct indexing but restricts the preview results (there are no spaces).

4.6 Summary

Bin Encoding has shown the feasibility of using encoding for providing privacy-preserving processing in the cloud. Privacy is achieved through the number of possible mappings and by using lossy encoding. The possibility of frequency attacks and the lack of randomness means Bin Encoding is not as strong as encryption-based searching schemes such as [140]. However, with the additional functionality and low overheads–similar to plain text–it gives Bin Encoding a good balance between security, performance and utility. Even greater privacy can be achieved using a distributed system, and it has been shown that distributed encoding could be a feasible method for protecting privacy in the cloud while providing practical performance. Naturally the performance of Bin Encoding will depend on the index, allowed leakages [140], and the number of documents in the system, but the simplicity of the scheme gives it potential. This chapter started to answer Hypothesis 3, where the following chapters will further explore the concept of distributed encoding.

Part II

FRIBs: Fragmenting Individual Bits

5

Scheme Model

In Chapter 4, the idea of using encoding to protect data was explored; however, Bin Encoding only supports string searching and cannot guarantee perfect accuracy. Therefore, another solution was required for researching Hypothesis 3. That scheme is a distributed encoding scheme, designed to support arbitrary computation with the goal of being both privacy-preserving and practical. The designed encoding scheme also provides a technique to address an issue with multi-party computation, where garbled circuits should not be reused. The scheme–codenamed Fragmenting Individual Bits (FRIBs)–is discussed for the remaining of this thesis, with the system model presented in this chapter.

5.1 Mathematical Definitions

For a greater understanding of the FRIBs scheme, two mathematical models were used: Petri Nets and Pi-Calculus. The standard model for Petri Nets is used with transitions, places, and tokens, as shown in Figure 5.1. A transition cannot occur unless inputting places have a token. When a transition occurs,

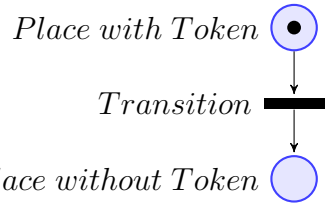

Figure 5.1: Petri Net definition

Table 5.1: Pi-Calculus definitions

Expression	Meaning
$\overline{c} < x >$	Send variable x through channel c.
$c(x)$	Receive and bind to variable x from channel c.
$A.B$	Run A, then run B.
$A\|B$	Run A and B in parallel.
$A.B\|C.D$	Run A, then run B and C in parallel, then run D.
$A(c) = c(x)$	Define A with channel c as $c(x)$.
$(k, l) \xrightarrow{map} (x)$	Use key k in lookup table l, binding the output to x.
$(x) \xrightarrow{ran(r)} (y)$	Perform a randomising operation using r.
$(x) \xrightarrow{rstates} (..., x, ...)$	Hide x in an array of random possible values for x.
$(x) \xrightarrow{operations} (y)$	Some operations are applied to x, resulting in y.

all input places lose a token, and all output places receive one token. This model was used to show the data transfers between servers and the concurrency, where each server needs to be running at the same time.

Pi-Calculus is the other mathematical representation used in this thesis. Petri Nets provide an abstract view of data flow between servers, whereas Pi-Calculus shows a more advanced process model and an idea of the operations occurring. Some customised expressions were used to help give the abstract data flow of FRIBs, given in Table 5.1, where the last four rows are used to provide more context into the workings of FRIBs.

5.2 Overview

The FRIBs scheme has been designed to distribute each individual bit across many service providers, while still allowing arbitrary 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 [23]. 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 privately by building functions from logic gates.

Another example is given in Figure 5.2, where we say a bit is a puzzle and

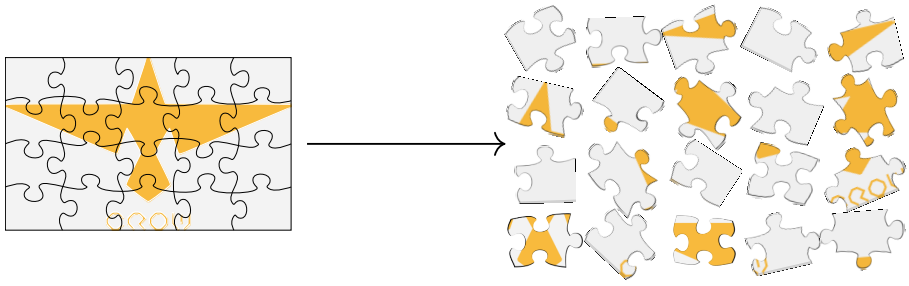

Figure 5.2: A bit can be seen as a puzzle, where all the pieces are needed to reconstruct the bit. In this example some information about the puzzle (bit) can be obtained from some of the pieces (fragments)

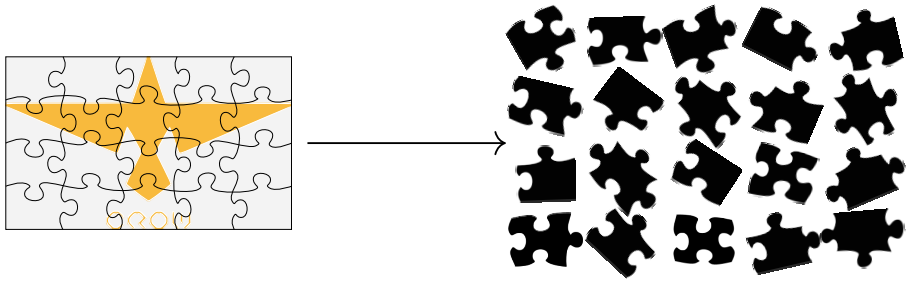

Figure 5.3: In this example the lights are turned out, so the pieces are not visible, meaning no information can be obtained from them

the fragments are the pieces [167]. Once fragmented, the puzzle can only be recreated by putting the pieces together in the correct order. However, in this example, one can learn something about the puzzle/bit by looking at a piece, whereas in Figure 5.3 the lights are turned off, meaning no information can be learnt about what the puzzle represents without putting the pieces together, which turns on the light. The concept of fragmentation is further explained in Section 5.3.1; however, if the fragmentation is all XOR operations ($bit = F_A \oplus F_B \oplus F_C$), then the last fragment still affects the result (lights out), where with all AND operations, the result could be revealed by a 0.

Privacy from a single entity is achieved by the distributed fragments being stored and processed on different servers hosted by different entities.[1] These distributed fragments are not a subset of the data in a way used by traditional

[1]For example, using Google, Microsoft and Amazon, such that no single entity has access to the others.

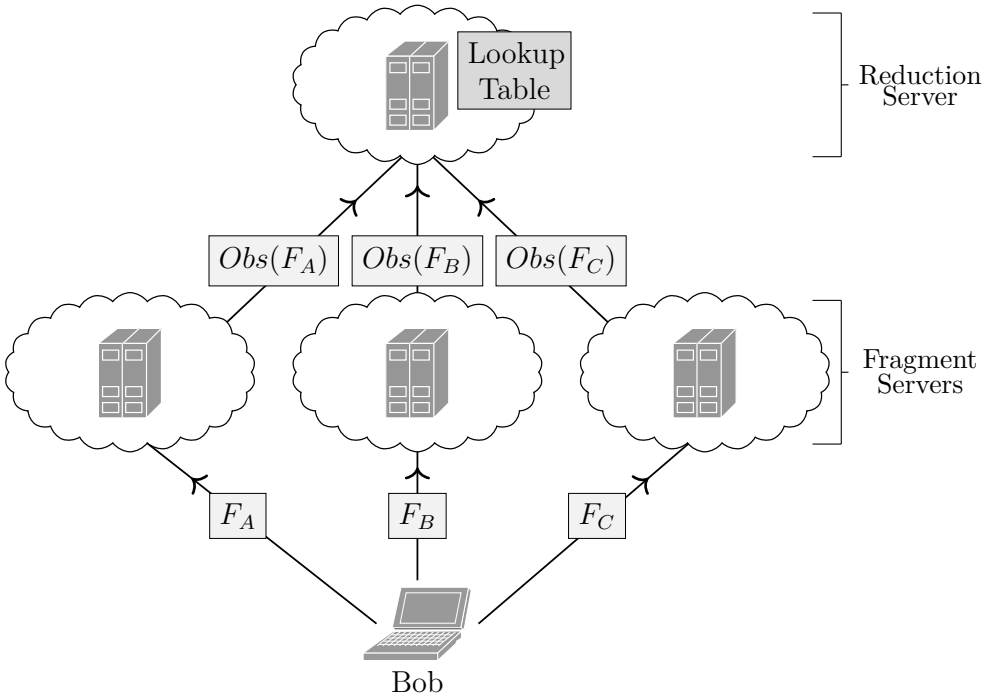

Figure 5.4: System model with three fragment servers and a reduction server

distributed systems, as each fragment holds a segment of a single bit which by itself represents nothing–FRIBs is not a divide and conquer approach. One example is a database table split over multiple servers [168]; each holds a different piece of information or subset, but a fragment holds part of a bit, meaning each server holds part of all the information. Therefore, processing occurs over all the fragments on each server with communication between them, requiring each server to run the same function/program at the same time. FRIBs can be seen as a subset of secure multi-party computation, but is a simpler technique, as encryption is only required for network communication. The benefit of FRIBs over current secure multi-party computation schemes is how FRIBs builds garbled circuits and that fragments (known as shares in multi-party computation) are not encrypted. This will be explored further in this chapter.

5.3 System Model

As data is being uploaded, each bit is split into fragments–F_A, F_B and F_C–in Figure 5.4, where *Obs* is an obfuscation function. Each fragment is protected using public-key cryptography[2] and sent to a separate fragment server, managed by different cloud providers and organisations. When these fragments are stored on the servers, they can be encrypted when at rest[3], but the keys are managed by the cloud. Depending on the fragmentation algorithm used, each fragment server can learn all of the bits, through to zero (the ideal solution) as touched upon with Figures 5.2 and 5.3.

5.3.1 Fragmentation

The fragmentation algorithm defines how the fragments are joined together to get the result bit. Keeping the fragmentation algorithm private can be seen as a type of secret; however, there are a limited number of algorithms available, for example when using two logic operations $\circledcirc_0$ and $\circledcirc_1$, where $\circledcirc_0, \circledcirc_1 \in \{\oplus, \vee, \downarrow, \wedge, \overline{\wedge}\}$[4], and with three fragment servers such that $F_A \circledcirc_0 F_B \circledcirc_1 F_C = bit_x$ gives less than 25 possible combinations. Therefore, in this thesis we will say the fragmentation algorithm is known.

The fragmentation algorithm can leak bits if it is known by the fragment servers; setting $\circledcirc_0 = \circledcirc_1 = \wedge$ (AND logic gates) reveals information if the fragment server receives a 0, because the result must therefore be *low*. Randomising the fragments gives Table 5.2, where only one entry gives a *high* result value. Assuming an equal probability for receiving a *low* or *high* bit, a fragment server could learn approximately 28% of the bits if the fragmentation operations are known.[5] Limiting the number of 0 fragments to one, as shown in Table 5.3, reduces this to approximately 17%. For a 32-bit value, this is only five bits, whereas using $\circledcirc_0 = \circledcirc_1 = \oplus$ means no bits are leaked to the fragment servers even if $\circledcirc_0$ and $\circledcirc_1$ are known. Therefore, this is the fragmentation algorithm used in this thesis, with any analysis assuming $\circledcirc_0$ and $\circledcirc_1$ are known.

[2]Because any communication between servers, and/or the client should be encrypted
[3]Data at rest is when data is in a stored state, i.e. not being processed or in transit
[4]Symbol definitions: $\oplus$ =XOR, $\vee$ =OR, $\downarrow$=NOR, $\wedge$ =AND, $\overline{\wedge}$ =NAND
[5]Calculated by $\frac{1}{2}$ of bits *low*, and $\frac{4}{7}$ fragments are 0 if the fragment is *low*.

Table 5.2: All options for fragment values.

Value	$\mathbf{F_A}$	$\mathbf{F_B}$	$\mathbf{F_C}$
low	0	0	0
low	0	0	1
low	0	1	0
low	0	1	1
low	1	0	0
low	1	0	1
low	1	1	0
high	1	1	1

Table 5.3: Reduced options for fragment values.

Value	$\mathbf{F_A}$	$\mathbf{F_B}$	$\mathbf{F_C}$
low	0	1	1
low	1	0	1
low	1	1	0
high	1	1	1

An example of encoding an 8-bit value 01101101_2, where $\circledcirc_0 = \circledcirc_1 = \oplus$ is given as $F_A = 01010011$, $F_B = 11011001$ and $F_C = 11100111$. Fragments F_A and F_B can be random, where F_C is required to make $F_A \oplus F_B \oplus F_C$ equal the bit value. The fragments F_A, F_B, and F_C are encrypted for transmission to their respective fragment server. This encryption will be part the secure connection from the client to the server; for example, transport layer security.

5.3.2 Fragment Servers

The fragment servers store fragments and execute operations over these fragments. Note that a 32-bit plain-text value is comparable to 32 fragments (where the fragments are from a 32-bit value), meaning at rest the size of the encrypted data stored is the same as if it were plain text ($size(E(value)) == size(E(fragments))$). Programs are executed over the fragments by concatenating fragments together, which later have the operation applied at the reduction stage. These concatenated values or fragment states need to maintain the order of operations; an operation over two fragments will be denoted by 0, where *low* fragments are represented as 11 during processing: for example, the fragment 01010011_2 is represented as $< 11, 1, 11, 1, 11, 11, 1, 1 >$.

Applying an operation (NAND, OR, or any an arbitrary bit operation) to the set of fragments $F_0 =< 11, 1, 11, 1, 11, 11, 1, 1 >$ and $F_1 =< 1, 1, 1, 11, 11, 1, 11, 11 >$, gives the resulting fragment $F_3 =< 1101, 101, 1101, 1011, 11011, 1101, 1011, 1011 >$. Applying another operation to F_3 and itself (F_3), gives $F_4 =<$ 1101001101, 10100101, 1101001101, 1011001011, 110110011011, 1101001101, 1011001011,

1011001011 >. In this case, the concatenation needed to use 00 to maintain state; for example, 1101001101 is $(0 \circledcirc 1) \circledcirc (0 \circledcirc 1)$ where $\circledcirc$ represents the operation. Note that once an operation has been applied, the same operation must be applied until the fragment is reduced back to a single bit.[6]

5.3.3 Reduction Server

As more fragments are concatenated together, the resulting fragment states will continue to grow. At a predefined point in the program, the fragments are reduced, which in turn applies the operation as it reduces the fragment back to a single bit. The point where a reduction occurs can be hardcoded into the program or added by a compiler, but reduction is needed once the fragment states are known to be at a certain size. The program is the same across all fragment servers; therefore, each server performs a reduction at the same time, because reducing a fragment requires information about all fragments for the operation to be computed correctly. At this point, we will say a separate server–known as the reduction server–is used, where all N fragment servers send their fragments to (over a secure medium) during the reduction step. Once the reduction server has received each fragment, it uses a precomputed Lookup Table (LUT) to get the reduced fragments to send back to each fragment server, thus, applying the operation. However, if each fragment was sent and returned from the reduction server in the current format (1011 for example), then some of the data could be decoded with a known fragmentation algorithm.

Since the reduction server is performing a simple lookup, we can obfuscate each fragment to a unique value, like a garbled circuit. For example, each server can hash the fragment with a server unique salt value or use a random mapping. Now, the reduction server should not know the state of the fragments it has received.[7] Protecting the reduced fragments is slightly more difficult. A public-key cryptography scheme can be applied to each reduced fragment, such that only the single server can decrypt the fragment.[8] The LUT is built offline and sent to the reduction server; therefore, the reduction server only receives a

[6] This is for the universal case, where the reduction could be defined such that 110101101 is $(0 \oplus 1) \wedge (0 \oplus 1)$.

[7] At this point, patterns can be seen if states are the same; this is addressed in Section 5.4.3

[8] The need to encrypt the resulting fragments is removed in Section 5.4.

protected lookup table, and all the reduced fragments are already encrypted. Another security benefit given is that each public key for the individual servers and their reduced fragments can remain private from the reduction server (which would also allow a symmetric encryption scheme to be used).

For example, when reducing the non-obfuscated fragments F_A, F_B and F_C below, each index into the fragment vectors is reduced; therefore, each index is treated separately.

$$F_A =< 1101, 101, 1101, 1011, 11011, 1101, 1011, 1011 >$$
$$F_B =< 11011, 1101, 1101, 1011, 1011, 1011, 11011, 1101 >$$
$$F_C =< 101, 1011, 11011, 1101, 101, 1101, 101, 11011 >$$

The result fragments (which are not encrypted in this example) are $F_A =< 0, 0, 1, 1, 0, 1, 0, 0 >$, $F_B =< 1, 1, 0, 0, 0, 1, 0, 0 >$ and $F_C =< 0, 0, 0, 0, 1, 1, 1, 0 >$. The 8-bit values for this example are $a = 133$ and $b = 27$ where $a \barwedge b = 254$, as can be solved via $F_A \oplus F_B \oplus F_C = 11111110_2$.

One improvement to reduce the usability of patterns observed by the reduction server is to use multiple reduction servers. For example, two reduction servers could be set up where the fragment servers alternate reduction requests between the two, as shown in Figure 5.5. The lookup tables should be unique, using different keys and obfuscated values. A pseudo-random number generator can be used to define which reduction server to send the requests too, where all the fragment servers use the same seed value.

5.3.4 Current Mathematical Models

To further explain the reduction server model of the FRIBs scheme, a Petri Net is given in Figure 5.6. This defines the model with two fragment servers and one reduction server. Places P_1, P_2 and P_3 are running on the first fragment server, with places P_4, P_5, and P_6 on the second fragment server, where places P_7 and P_8 are on the reduction server. The transition *Process* is the algorithm computing over the data, which combines fragments until a fixed point in the execution, with both *Process* transitions computing the same operations. Once the fragments need reducing, the transition *Reduction* is reached. This transition obfuscates the fragment and sends it to the reduction server. Once

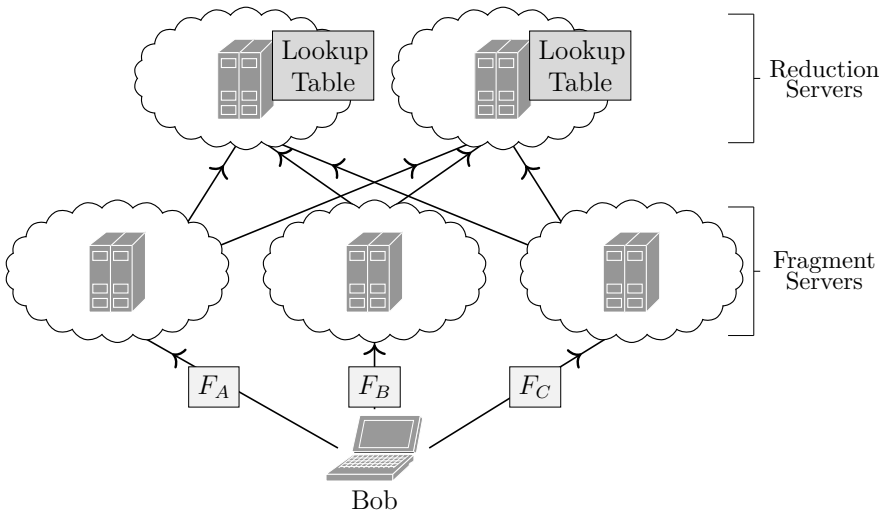

Figure 5.5: System model with three fragment servers and two reduction servers

the reduction server has received both obfuscated states, the *Lookup* transition can get the obfuscated result fragments for each fragment server. When a fragment server receives its reduced fragment, computation can continue.

Expanding on the Petri Net model, Pi-Calculus will be used to define the reduction server model in a dynamic manner. The lookup operation is defined in Equation 5.1, which takes the LUT, and the channel *key*. Note that the key can be made up of many states, and can return multiple states. It is used to obfuscate a state on a fragment server, and for the reduction server to get the result fragments.

$$
\begin{aligned}
&Lookup(table\_name, key) = \\
&\quad key(state_0,\ \dots\ , state_x, result). \\
&\quad (state_0,\ \dots\ , state_x, table\_name) \xrightarrow{map} (new\_state_0,\ \dots\ , new\_state_x). \\
&\quad \overline{result} < new\_state_0,\ \dots\ , new\_state_x > . \\
&\quad Lookup(table\_name, key)
\end{aligned}
\tag{5.1}
$$

A reduction server waits for the obfuscated values, using them to get the new states, and returns them to the fragment servers, shown in Equation 5.3.

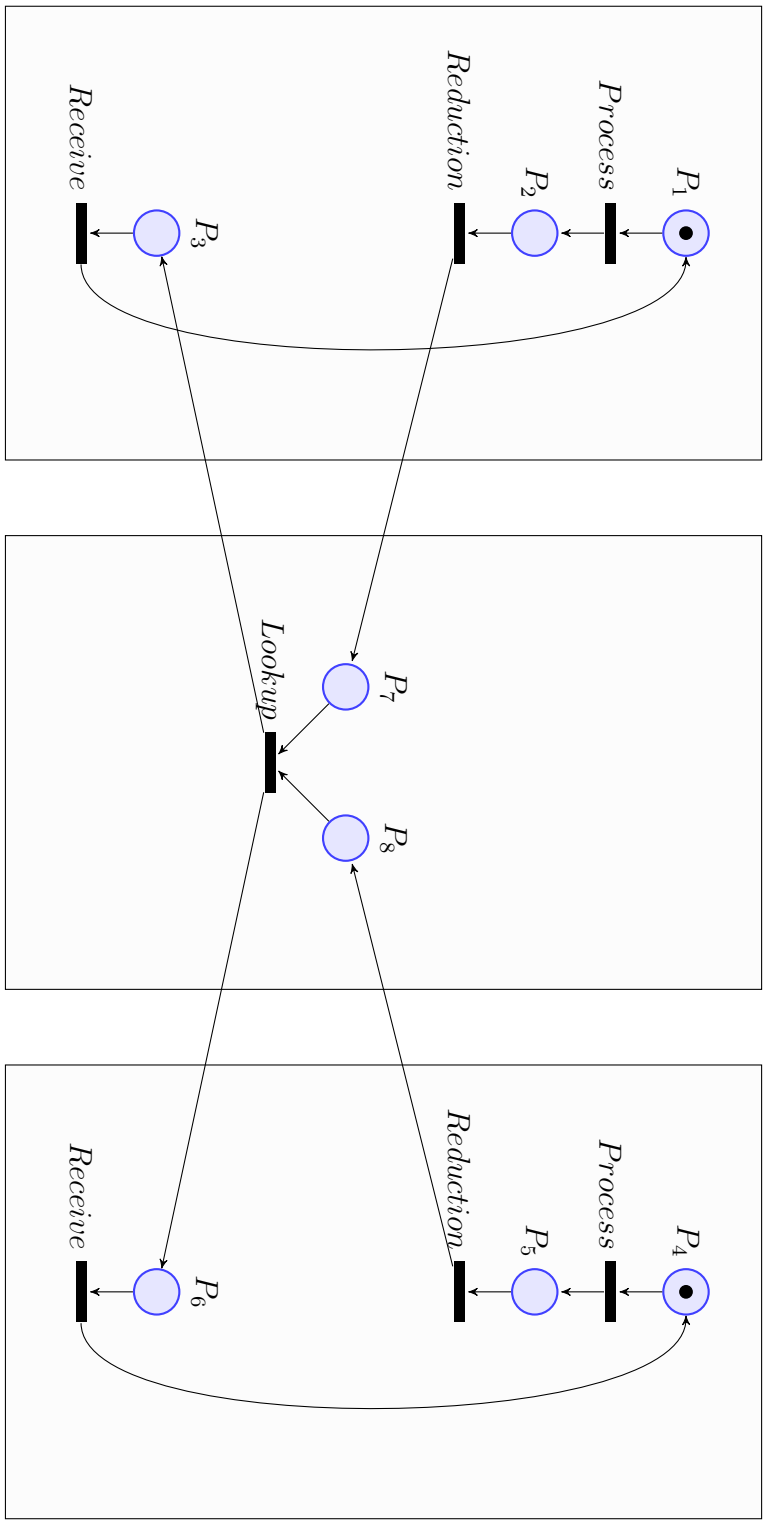

Figure 5.6: Two fragment servers and a reduction server Petri Net model

$$
\begin{aligned}
&Reduction(server\_frag_0,\ \ldots\ , server\_frag_x, table) = \\
&\quad server\_frag_0(state_0, c_0)|\ \ldots\ |server\_frag_x(state_x, c_x). \\
&\quad \overline{table} < state_0,\ \ldots\ , state_x, tc > .tc(new\_state_0,\ \ldots\ , new\_state_x). \\
&\quad \overline{c_0} < new\_state_0 > |\ \ldots\ |\overline{c_x} < new\_state_x > . \\
&\quad Reduction(server\_frag_0,\ \ldots\ , server\_frag_x, table)
\end{aligned}
\tag{5.2}
$$

$$
\begin{aligned}
&ReductionServer(server\_frag_0,\ \ldots\ , server\_frag_x, table\_name) = \\
&\quad Lookup(table\_name, table)\ | \\
&\quad Reduction(server\_frag_0,\ \ldots\ , server\_frag_x, table)
\end{aligned}
\tag{5.3}
$$

The fragment servers are made of two parts: Equations 5.4 and 5.5. Equation 5.4 handles obfuscating the state, sending it to the reduction server, and returning the result, where Equation 5.5 is the program being executed; this will be the same program for each fragment server, where some states are combined and then reduced. An example fragment server is given in Equation 5.6.

$$
\begin{aligned}
&Reduce(server\_red, table, state\_in) = \\
&\quad state\_in(state, state\_out).\overline{table} < state, c > .c(obf\_state). \\
&\quad \overline{server\_red} < obf\_state, rc > .rc(new\_state). \\
&\quad \overline{state\_out} < new\_state > . \\
&\quad Reduce(server\_red, table, state\_in)
\end{aligned}
\tag{5.4}
$$

$$
\begin{aligned}
&Program(reduce) = \\
&\quad state \xrightarrow{operations} new\_state.\overline{reduce} < new\_state, o > .o(state)
\end{aligned}
\tag{5.5}
$$

$$
\begin{aligned}
&FragmentServer(table\_name, server\_red) = \\
&\quad Lookup(table\_name, table) \mid Reduce(server_{red}, table, rstate) \mid \\
&\quad Program(rstate)
\end{aligned}
\tag{5.6}
$$

Sample configuration is given in Equation 5.7, with two fragment servers and one reduction server. The table names represent that a different LUT is being used, and was constructed offline.

$$
\begin{aligned}
&Compute = (new \;\; server_1, server_2) \\
&\quad ReductionServer(server_1, server_2, \text{“}reduction\_table''\text{)} \mid \\
&\quad FragmentServer(\text{“}frag\_1\_table'', server_1) \mid \\
&\quad FragmentServer(\text{“}frag\_2\_table'', server_2)
\end{aligned}
\tag{5.7}
$$

5.4 Removing the Reduction Server

Currently, the system model for FRIBs is similar to that of multi-party computation schemes where garbled circuits are used. The difference is that the reduction server has no knowledge of the data or the program, and can only learn patterns based on reusing of LUTs (hence why other multi-party computation schemes also cannot reuse garbled circuits). However, instead of using a singular LUT which contains the reduced fragments, separate LUTs can be used such that each result only reveals one fragment servers reduced value.

5.4.1 Performance Orientated Model

With the reduction server model, each fragment server sends their obfuscated state and waits for the reply, as shown in Figure 5.7. Therefore, the Round Trip Time (RTT) will be responsible for the majority of the computation time. This can be halved by removing the need for the fragment servers to wait for a response. To achieve this, each fragment server can be its own reduction server, removing the need for encrypting or obfuscating the output states, as each LUT will only output its own result fragment value. This is shown in

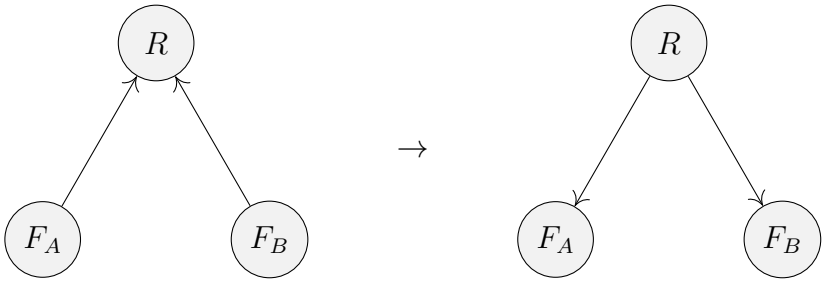

Figure 5.7: Reduction server model, where fragment servers wait for the response

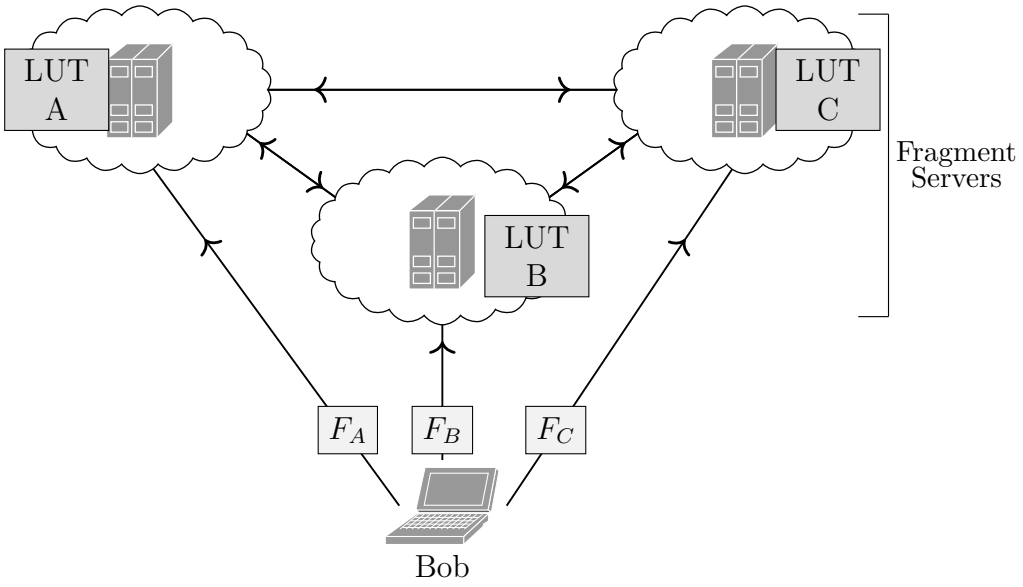

Figure 5.8: System model with three fragment servers and no reduction servers

Figure 5.8, where the reduction server is removed and each fragment server has a unique LUT. During the reduction step, each fragment server sends an obfuscated state[9] to the other fragment servers, as shown in Figure 5.9.

Figure 5.10 gives a dynamic Petri Net model, where the fragment server waits for all obfuscated states before being able to get its reduced fragment value during the *Receive* transition, which then allows processing to continue. Figure 5.11 presents the performance model with three servers. This shows that even if the servers are running at slightly different timings, they all must wait until each is at the same stage of the program.

A slight redefinition of Equation 5.1 is given in Equation 5.8, where the

[9]The concatenated fragment is mapped to an obfuscated state value.

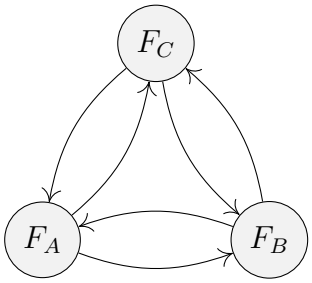

Figure 5.9: No reduction server model; the fragment servers send all at once

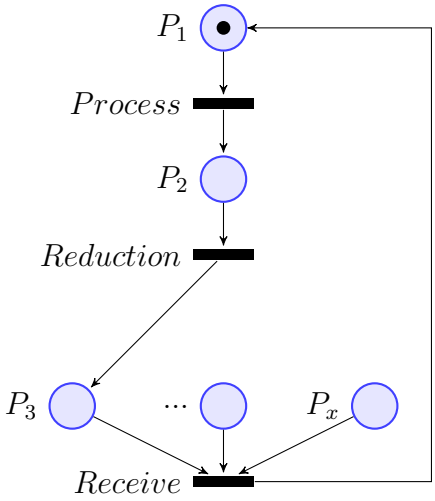

Figure 5.10: Variable server Petri Net for the FRIBs performance model

output is now a single result fragment value.

$$
\begin{aligned}
&Lookup(table\_name, key) = \\
&\quad key(state_0, \ \ldots, state_x, result). \\
&\quad (state_0, \ \ldots, state_x, table\_name) \xrightarrow{map} (new\_state). \\
&\quad \overline{result} < new\_state > . \\
&\quad Lookup(table\_name, key)
\end{aligned}
\tag{5.8}
$$

A new process is defined in Equation 5.9, which obfuscates the servers' state. The obfuscated state is then sent to one of the other servers. The process waits to receive the other servers obfuscated state and returns it.

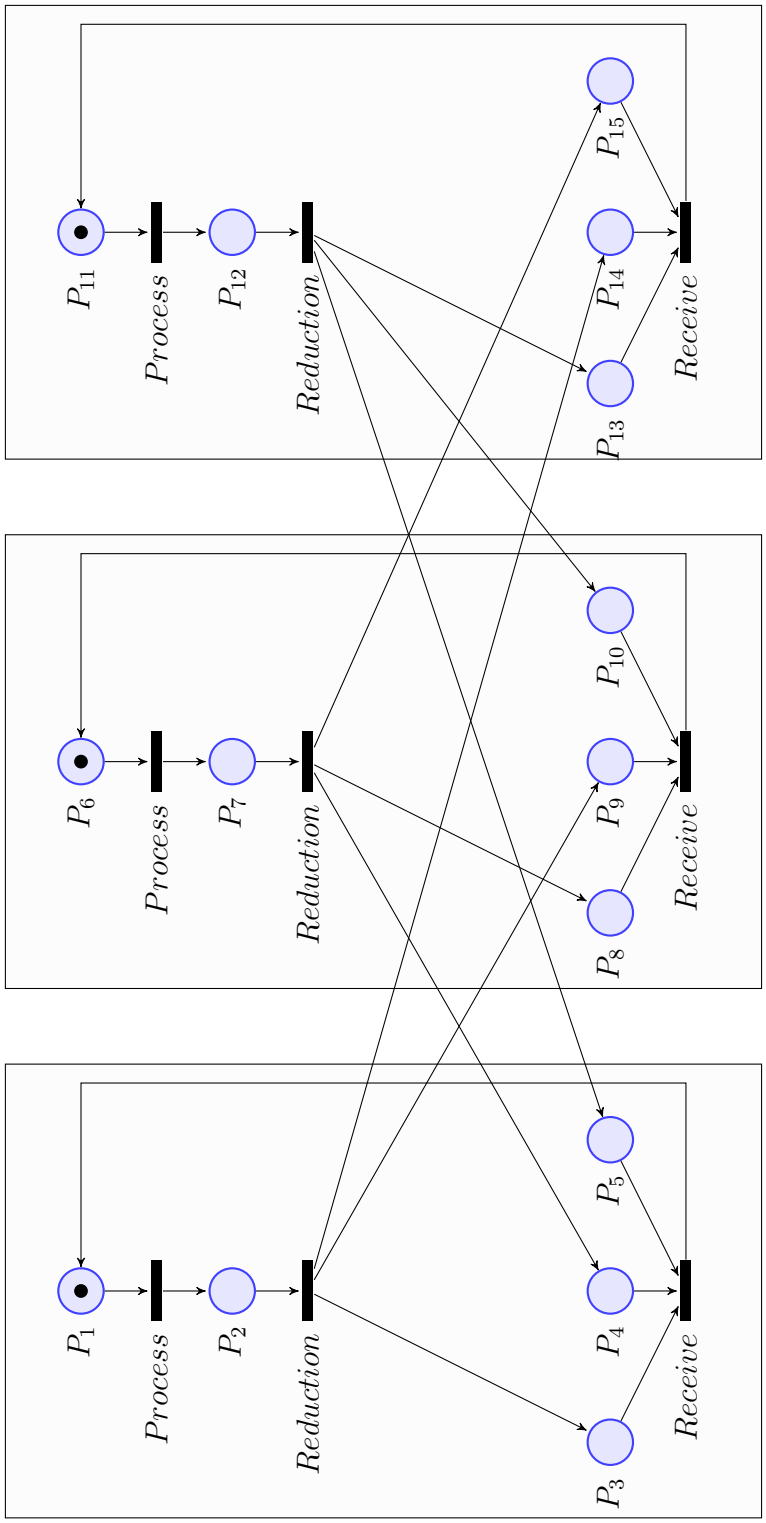

Figure 5.11: Three server Petri Net for the FRIBs performance model

$$
\begin{aligned}
&SubReduce(server\_in, server\_out, table, state\_in) = \\
&\quad state\_in(state, state\_out).\overline{table} < state, c > .c(new\_state). \\
&\quad \overline{server\_out} < new\_state > .server\_in(r\_state). \\
&\quad \overline{state\_out} < r\_state > . \\
&\quad SubReduce(server\_in, server\_out, table, state\_in)
\end{aligned}
\tag{5.9}
$$

The process *Reduce* is redefined such that its parameters are now channels to a *SubReduce* process, for example sr_1. It still waits to receive the current state through *state_in* and sends it to each *SubReduce* process. Upon receiving the obfuscated values of the other servers, it uses the LUT to get its new state.

$$
\begin{aligned}
&Reduce(sr_1,\ ...\ , sr_x, table, state\_in) = \\
&\quad state\_in(state, result).(\overline{sr_1} < state, s_1 > .s_1(state_1) \mid ... \mid \\
&\quad \overline{sr_x} < state, s_x > .s_1(state_x)).\overline{table} < state, state_1,\ ...\ ,\ state_x, s > . \\
&\quad s(new\_state).\overline{result} < new\_state > . \\
&\quad Reduce(sr_1,\ ...\ , sr_x, table, state\_in)
\end{aligned}
\tag{5.10}
$$

Therefore, a fragment server must now be redefined, and is given in Equation 5.11. An example configuration is given with three fragment servers in Equation 5.12.

$$
\begin{aligned}
&FragmentServer(server_{1-0}, server_{0-1},\ ...\ , server_{x-0}, server_{0-x}) = \\
&\quad Lookup(\text{“}result\_table'', table_r) \mid \\
&\quad Lookup(\text{“}table1'', table_1) \mid ... \mid Lookup(\text{“}tableX'', table_x) \mid \\
&\quad SubReduce(server_{1-0}, server_{0-1}, table_1, state_1) \mid ... \mid \\
&\quad SubReduce(server_{x-0}, server_{0-x}, table_x, state_x) \mid \\
&\quad Reduce(state_0,\ ...\ , state_x, table_r, rstate) \mid \\
&\quad Program(rstate)
\end{aligned}
\tag{5.11}
$$

$$
\begin{aligned}
&Compute = \\
&\quad FragmentServer(server_{1-0}, server_{0-1}, server_{2-0}, server_{0-2}) \mid \\
&\quad FragmentServer(server_{0-1}, server_{1-0}, server_{2-1}, server_{1-2}) \mid \\
&\quad FragmentServer(server_{0-2}, server_{2-0}, server_{1-2}, server_{2-1})
\end{aligned}
\tag{5.12}
$$

An expanded version of Equation 5.12 is presented in Equation 5.13, showing the differing LUTs more clearly.

$$
\begin{aligned}
&Compute = \\
&\quad Lookup(\text{“}table0'', table_0) \mid Lookup(\text{“}table1'', table_1) \mid \\
&\quad Lookup(\text{“}table2'', table_2) \mid Lookup(\text{“}table0-1'', table_{0-1}) \mid \\
&\quad Lookup(\text{“}table0-2'', table_{0-2}) \mid Lookup(\text{“}table1-0'', table_{1-0}) \mid \\
&\quad Lookup(\text{“}table1-2'', table_{1-2}) \mid Lookup(\text{“}table2-0'', table_{2-0}) \mid \\
&\quad Lookup(\text{“}table2-1'', table_{2-1}) \mid \\
&\quad SubReduce(server_{1-0}, server_{0-1}, table_{0-1}, state_{0-1}) \mid \\
&\quad SubReduce(server_{2-0}, server_{0-2}, table_{0-2}, state_{0-2}) \mid \\
&\quad SubReduce(server_{0-1}, server_{1-0}, table_{1-0}, state_{1-0}) \mid \\
&\quad SubReduce(server_{2-1}, server_{1-2}, table_{1-2}, state_{1-2}) \mid \\
&\quad SubReduce(server_{0-2}, server_{2-0}, table_{2-0}, state_{2-0}) \mid \\
&\quad SubReduce(server_{1-2}, server_{2-1}, table_{2-1}, state_{2-1}) \mid \\
&\quad Reduce(state_{0-1}, state_{0-2}, table_0, rs_0) \mid \\
&\quad Reduce(state_{1-0}, state_{1-2}, table_1, rs_1) \mid \\
&\quad Reduce(state_{2-0}, state_{2-1}, table_2, rs_2) \mid \\
&\quad Program(rs_0) | Program(rs_1) | Program(rs_2)
\end{aligned}
\tag{5.13}
$$

5.4.2 Privacy Orientated

The limitation of the performance orientated design is that each fragment server knows if another is in the same state or not, allowing for patterns to be observed. A simple method to reduce the observability would be to have mul-

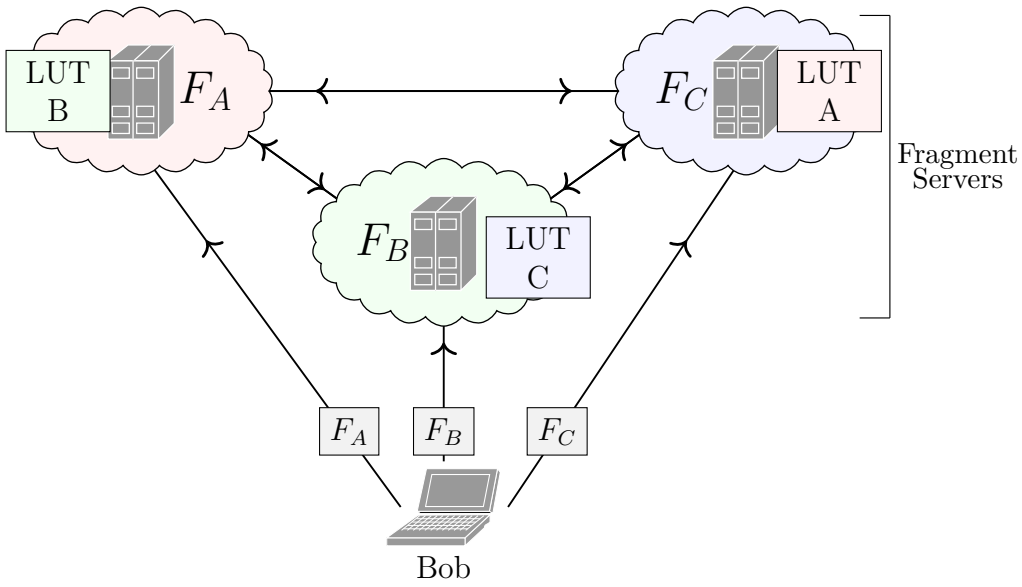

Figure 5.12: System model with three fragment servers, where the LUTs are rotated

tiple obfuscated states for the same real state (concatenated fragment value). However, this would increase the size of the LUTs, as more combinations are required. Depending on size requirements, this is something that should be done regardless, as it will limit any pattern learning over each state having a single entry. This method can also be used if a state is being reached more often than others, as the number of obfuscated states can be a ratio of the number of times it is thought to be reached.

The knowledge learnt about other fragment servers' states can be minimised further, such that a fragment server only knows if all the other servers are in the same state. Therefore, if only one other fragment server changes state, the fragment server gains no knowledge to which other fragment server changed state (it could be more than one). This method will be explained for a system with three fragment servers. Instead of a fragment server storing its own result LUT, the LUTs are rotated such that each server has another server's result LUT, as shown in Figure 5.12. On reduction, instead of a fragment server sending two obfuscated values to the other two servers (one to each), the possible result output values (becomes a smaller LUT) are sent to the fragment server which the LUT belongs to, and an index for the other server.

The smaller LUT only has two keys, which are indexes, because the fragment server storing the LUT only selects the rows based on its own state. When the fragment server receives its small LUT, it can select the row based on its own state, then uses the other server's index to get the output value. The Petri Net for this model is still the same as Figure 5.11, except the values sent are different.

Currently, this still reveals if any state remains the same. To hide individual states, the two other fragment servers have a pseudo-random number generator (for this fragment server) using the same seed.[10] At each reduction step, the generator is used to change the values. Each row in the smaller LUT can be XORed with r_0, where r_0 is a random number. The rows can then be bit shifted[11] by another random number r_1. The other fragment server has the same values r_0 and r_1, and can use r_1 to shift the index value it sends, and use r_0 to determine if the result bit needs to be flipped. When a fragment server receives both the now obfuscated LUT and index (plus a flip value), it still uses its state to determine the row in the LUT. The issue remaining is that if all the other fragment servers' states remain the same, the column in the LUT could reveal the two other states are the same.

5.4.3 Enhanced Privacy Model

To hide the fact the other states are the same can be achieved, but this will require additional latency. The same model as Section 5.4.2 is used, with a step added before a server receives the obfuscated LUT. A server must first ask for a subset of rows–in random order–it wants in the obfuscated LUT. Therefore, instead of the obfuscated LUT containing all possible states for the server, it only has a subset (now known as a small obfuscated LUT). This also reduces the amount of data being sent over the network. By enforcing varying subsets, the column approach[12] to see if the other servers' states remained the same cannot work as efficiently (if at all). The enforcement of varying subsets can be done by the server creating the small obfuscated LUT, by refusing to

[10] The user can set the seed values, or the fragment servers could securely agree/generate a new seed at given intervals.

[11] A shuffle function could be used as well, where the random number defines how the shuffle should occur.

[12] Where the column of the result state in the obfuscated LUT will always remain the same.

send the same rows if some condition is met, such as two requests for the exact same rows.

The final step is to hide all of the possible outputs of the small obfuscated LUT apart from the resulting reduced state. To achieve this, each row in the small obfuscated LUT now uses different random values for r_0 and r_1 (recall that r_0 is XORed over the row, and r_1 is a bit shift index). The fragment server sending the index and whether the bit needs to be flipped now receives an index number for the correct row in the small obfuscated LUT, which it uses to get the correct r_0 and r_1 values. Note that this index number is not a state value, and will change even if the same state value is being reduced. So now we have two transfers (one RTT) between each fragment server, represented in Figure 5.13. Note that the flow will be shown visually in Section 5.5.1, which will help explain this concept.

At this point, the small obfuscated LUT is protected, where the random values could be thought of as secret keys. However, Hypothesis 3 requires that only encoding is to be used for achieving privacy-preserving computation. Classifying the simple XOR and rotation functions as a form of encryption is subjective. With 64 possible states for each reduction, the XOR value will be between 0 and 63, while the rotate value will be between 0 and 7. With so few possibilities, trying all combinations requires little effort–knowing which is correct is the challenge. Therefore, this technique can be considered a form of *Privacy-Preserving Encoding*, as it is too weak to be encryption.

Given the complexity of this model (known as the enhanced privacy model), the Pi-Calculus equations are defined for a three server model and are not dynamic. The same *Lookup* process can be used, as described in Equation 5.8. To generate the small obfuscated LUT, *SubLookup* is defined in Equation 5.14. This waits for the list of obfuscated states and creates the small obfuscated LUT accordingly. The process then waits to receive the row index i that this fragment server will use to get its reduced state. The value i is sent to the other server. It then applies a different random value r_0 and a random shift r_1 to each row using the pseudo-random number generator r_{gen} to the small obfuscated LUT. The small obfuscated LUT is sent to the other server, before waiting to receive the index value and bit flip value from that server, for row i of the small obfuscated LUT this server will receive.

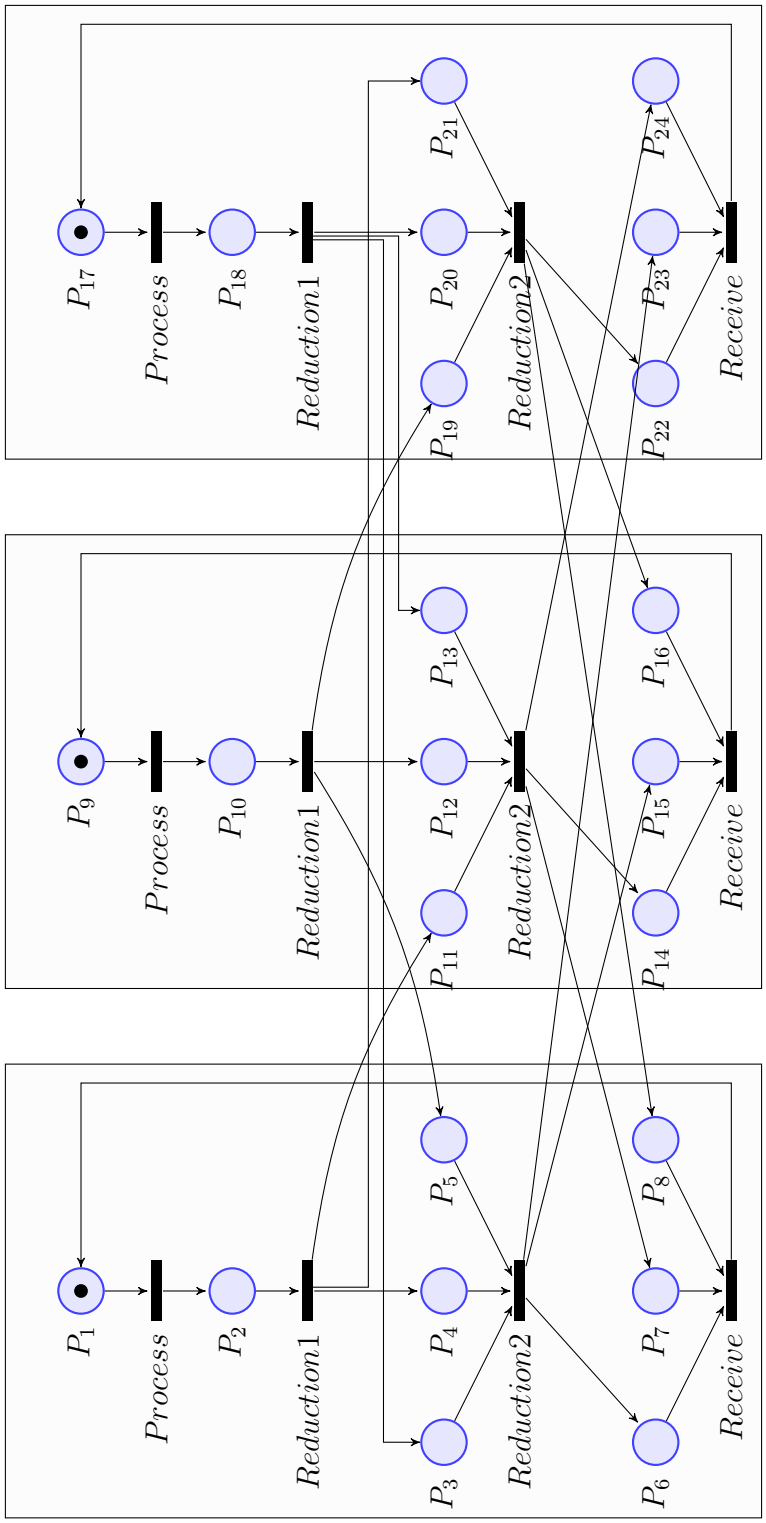

Figure 5.13: Three server Petri Net for the FRIBs enhanced privacy model

$$
\begin{aligned}
&SubLookup(table\_name, key, server\_in, server\_out, index,\\
&\quad r\_gen) = key(state, result)\ .\ server\_in(ostate_0,\ \ldots\ , ostate_x).\\
&\quad index(i)\ .\ \overline{server\_out} < i >\ .\\
&\quad (state, ostate_0,\ \ldots\ , ostate_x, table\_name) \xrightarrow{map} (sub\_table).\\
&\quad (sub\_table) \xrightarrow{ran(r_{gen})} (rsub\_table)\ .\\
&\quad \overline{server\_out} < rsub\_table >\ \mid server\_in(rindex, rflip).\\
&\quad \overline{result} < i, rindex, rflip >\ .\\
&\quad SubLookup(table\_name, key, server\_in, server\_out, r\_gen)
\end{aligned}
\tag{5.14}
$$

The opposite process of *SubLookup* is the new definition for *SubReduce* in Equation 5.15. This process obfuscates the server's state, and randomises it using r_{gen} and the row index ri used by the other server (row index ensures the right r_0 and r_1 values). The output is the index in the row and whether the bit needs to be flipped, which is sent to the other server through channel $server_{out}$. The second part of this process is to receive the small obfuscated LUT. First, it uses a different LUT to get a different obfuscated value, obf_state_0, and generates a random array of possible obfuscated states, including obf_state_0 at index i. The array is sent to the fragment server, which has the full LUT for this server (received in a *SubLookup* process). The small obfuscated LUT is received through channel $server_{in}$, and returned.

$$
\begin{aligned}
&SubReduce(server\_in, server\_out, table, index, state\_in, r\_gen) =\\
&\quad state\_in(state, result).\overline{table} < state, c_1 > .c_1(obf\_state).\\
&\quad (obf\_state) \xrightarrow{rstates} (i, ostate_0,\ \ldots\ , ostate_x).\\
&\quad \overline{server\_out} < ostate_0,\ \ldots\ , ostate_x >\ .\\
&\quad \overline{index} < i >\ .\ server\_in(ri)\ .\ (obf\_state) \xrightarrow{r_{gen}, ri} (rindex, rflip).\\
&\quad \overline{server\_out} < rindex, rflip >\ \mid server\_in(rsub\_table).\\
&\quad \overline{result} < rsub\_table >\ .\\
&\quad SubReduce(server\_in, server\_out, table, state\_in, r\_gen)
\end{aligned}
\tag{5.15}
$$

The *Reduce* process has three arguments, where $server_1$ is a channel to a *SubReduce* process, $server_2$ is a channel to a *SubLookup* process, and $state_{in}$ is the input/output channel for the process. Upon receiving a state to reduce, the state is sent to each server through either *SubReduce* or *SubLookup*. The values received are a row index, bit flip value, and a randomised small obfuscated LUT. Using the row index i which is the server's state, the reduced state can be retrieved by $subtable[i][rindex] \oplus rflip$.

$$
\begin{aligned}
&Reduce(server_1, server_2, state\_in) = \\
&\quad state\_in(state, result). \\
&\quad \overline{server_1} < state, tr_1 > | \overline{server_2} < state, tr_2 > . \\
&\quad tr_1(subtable) | tr_2(i, rindex, rflip). \\
&\quad (i, rindex, rflip, subtable) \xrightarrow{map} (new\_state). \\
&\quad \overline{result} < new\_state > . \\
&\quad Reduce(server_1, server_2, state\_in)
\end{aligned}
\tag{5.16}
$$

A fragment server can then be defined in Equation 5.17. It requires two LUTs to obfuscate the server's state for communicating with the other two servers using the *Lookup* process. A *SubLookup* and *SubReduce* process is needed to communicate with the other fragment servers. Finally, the *Reduce* and *Program* processes are required.

$$
\begin{aligned}
&FragmentServer(server_{1-0}, server_{0-1}, server_{2-0}, server_{0-2}, \\
&\quad table\_r\_name, table\_0-1\_name, table\_0-2\_name, r\_gen_1, \\
&\quad r\_gen_2) = \\
&\quad Lookup(table\_0-1\_name, table_{0-1}) \mid \\
&\quad Lookup(table\_0-2\_name, table_{0-2}) \mid \\
&\quad SubLookup(table\_r\_name, server_2, table_{0-2}, server_{0-2}, server_{2-0}, \\
&\quad index, r\_gen_2) \mid \\
&\quad SubReduce(server_{1-0}, server_{0-1}, table_{0-1}, index, server_1, r\_gen_1) \mid \\
&\quad Reduce(server_1, server_2, rstate) \mid Program(rstate)
\end{aligned}
\tag{5.17}
$$

Now, three fragment servers can be setup in parallel, as shown in Equation 5.18, to execute an arbitrary function/program over the fragmented data.

$$
\begin{aligned}
&Compute = \\
&\quad FragmentServer(server_{1-0}, server_{0-1}, server_{2-0}, server_{0-2}, \\
&\qquad \text{“}table2'', \text{“}table0-1'', \text{“}table0-2'', r\_gen_1, r\_gen_2) \mid \\
&\quad FragmentServer(server_{2-1}, server_{1-2}, server_{0-1}, server_{1-0}, \\
&\qquad \text{“}table0'', \text{“}table1-2'', \text{“}table1-0'', r\_gen_2, r\_gen_0) \mid \\
&\quad FragmentServer(server_{0-2}, server_{2-0}, server_{1-2}, server_{2-1}, \\
&\qquad \text{“}table1'', \text{“}table2-0'', \text{“}table2-1'', r\_gen_0, r\_gen_1)
\end{aligned}
\tag{5.18}
$$

During a reduction step, a fragment server only learns its new state, a list of possible states for one fragment server ($ostate_0$, ... , $ostate_x$), and no information about the other fragment server. It learns no information, because all it receives is an index into the correct row, and whether to flip the bit. This is not enough information to determine the server's state or if the state has changed.

5.5 Data Flow

5.5.1 Reduction

To help explain the enhanced privacy model in Section 5.4.3, Figure 5.14 (on page 106) gives a data flow diagram for a reduction step from the point of view of one fragment server. The steps are given below:

Step 1: The client has fragmented and distributed a single bit to the fragment servers, which are preparing for computation. Recall that 11 is a 0 when processing is occurring. Therefore, the bit is *high* ($1 \oplus 0 \oplus 0 = 1$).

Step 2: An operation is applied to the bit and itself (the bit). This operation could be a NAND function, and is shown as a 0 to give the current order of operations.

Step 3: A reduction stage has been reached, so each server converts the concatenated fragment to an obfuscated state value. Servers F_B and F_C have a unique random mapping for their concatenated fragment to server F_A. Server F_A also needs obfuscation for sending the list of states to F_C.

Step 4: Server F_A generates a random vector of states it would like included in its small obfuscated LUT. In this example, because there only four states, it asks for all four.

Step 5: Server F_A sends this vector to server F_C, which reveals nothing about its actual state. Server F_A also sends server F_B the index into the vector for the correct state. In this case, state 3 is at index 1. Server F_B also learns nothing, because it does not know the order of the vector.

Step 6: The result LUT for server F_A is stored on server F_C, meaning server F_C can get the possible resulting fragment values using its obfuscated state, which is the last index in this example. This is the small obfuscated LUT in plain form.

Step 7: The small obfuscated LUT is now rearranged to match the request vector. Both server F_B and F_C possess the same seed S_A for the pseudo-random number generator–F_A has no knowledge of the seed. Server F_C gets eight random values from this generator and applies an XOR and bit shift to each row in the small obfuscated LUT. The rotate could also be a shuffle function, so long as it is deterministic. Because server F_B has the same generator as F_C and knows the index of the correct row, it can get the random numbers it needs, in this case r_2 and r_3. It uses r_3 and its obfuscated state value 2 to set j to the correct index. Then it uses its obfuscated state and r_2 to determine if the result bit needs to be flipped or not, stored in f.

Step 8: Finally, the small obfuscated result LUT and values j and f are securely transmitted to server F_A, which can use them to get the result fragment value r.

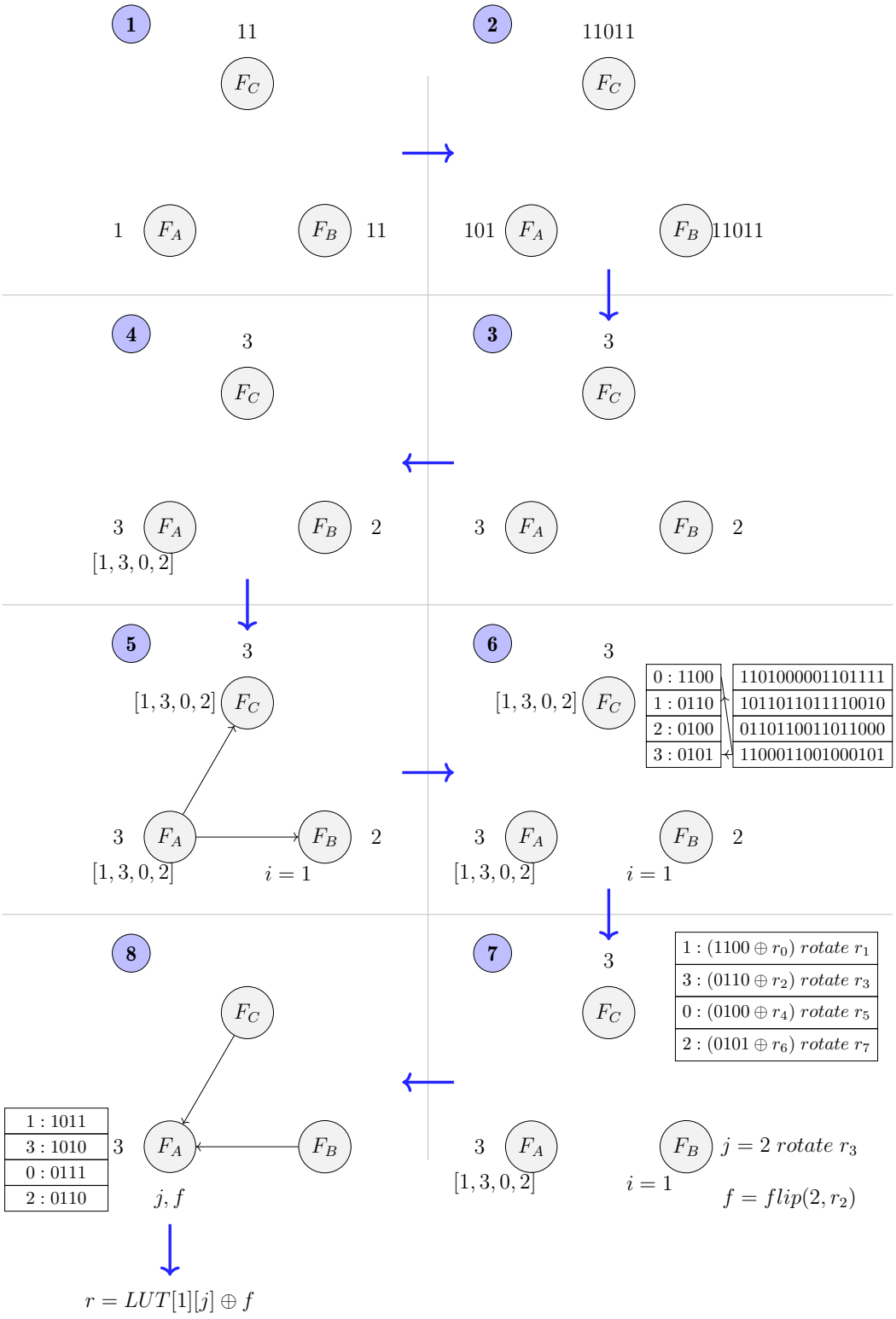

Figure 5.14: Reduction flow example for fragment server A

5.5.2 Securing Uploading and Downloading

There are many ways to transfer the fragments from the client to the fragment servers, with a few mentioned here.

Public-Key Cryptography

The most efficient method of uploading data to and from the fragment servers is with a public-key cryptosystem. Each fragment server can have its own private key and the public key of the user. Then, when the user wants to upload data, the fragments can be encrypted for each server. In terms of overhead, the client is only sending N times as much data as uploading to a single server, where N is the number of fragment servers. For example, a 32-bit integer split into three fragments would need 3×32-bit values encrypted. For downloading, the fragment server just encrypts the 32 fragments as a 32-bit integer using the client's public key. The client can decrypt each set of fragments and join them together to get the resulting 32-bit value.

Goldwasser-Micali

Depending on the size of the data being downloaded, each fragment server can use the XOR partially homomorphic encryption scheme by Goldwasser *et al.* to encrypt each fragment. For example, posting results on a publicly accessible web server, the web server can homomorphically join the fragments together. Clients with the decryption key can then download the encrypted bits. The disadvantage of using this scheme is the amount of memory/storage used, where a single bit becomes thousands.

The partially homomorphic encryption scheme can also be used for uploading data but requires a distributor server. For example, a client can encrypt a bit $E(bit)$ and send it to the distributor. Here, two random bits are encrypted $F_A = E(r_0)$ and $F_B = E(r_1)$, giving two fragments. The final fragment is $F_C = E(bit) \times E(r_0) \times E(r_1)$. These fragments can then be sent to the fragment servers. Another disadvantage appears where at least one fragment server would have the same decryption key as the original encrypted value $E(bit)$. Ultimately, using this partially homomorphic encryption scheme adds more overheads than positive properties, but is shown as a possibility.

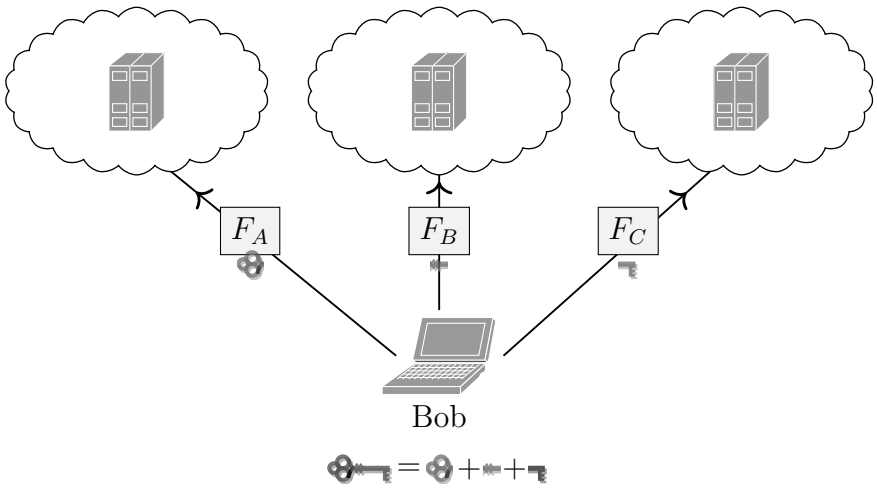

Figure 5.15: Splitting a public key across three fragment servers

Encryption within Fragments

With arbitrary processing available in the FRIBs scheme, encryption algorithms can be computed. The public key can be split into fragments like any other piece of data; an abstract example is given in Figure 5.15. When data is being downloaded, it can first be encrypted with the user's public key within the fragments. Therefore, even if the fragments are stolen, the data is still encrypted. Decryption can also be computed within the fragments. This allows data to be encrypted before storage and decrypted when needed, adding another layer of protection. The downside is that if all the fragment servers are compromised, so would the decryption key. Ideally, the decryption key would not be stored on the fragment servers, but only be used when a program is running over the data. Then, if a server is compromised, the malicious user would have to wait for the decryption key fragments on each server, adding more difficulty and time.

5.5.3 Fragment Data Channels

Data is sent between the fragment servers during processing. These channels need to be protected from eavesdropping to stop resulting states from being captured. Any encryption scheme can be used, but for performance, a symmetric scheme such as AES is recommended. For most implementations, it

would be possible to utilise a transport layer security library when setting up the connection.

5.6 Summary

Two core models have been presented for FRIBs in this chapter: one focused on performance, and the other on privacy, known as the performance model and the enhanced privacy model respectively.[13] The enhanced privacy model is the primary model, where a fragment server does not learn any information about states of the other fragment servers, and no patterns can be observed. This allows the LUTs to be reused, which is a challenge for most multi-party computation techniques. However, the reusability comes at the cost of the data not being protected by a secret key, like encryption, as retrieving all the fragments from each server will reveal the data. The other cost is each reduction request takes the largest RTT between all fragment servers, where the performance model halves this and sends less data over the network. The practicality can be estimated by network latency times, as lookup and processing requires little overhead. This means the fragment servers will spend most of their time waiting for network transfers, allowing for parallelisation to improve performance even with a single processor.

[13] The model using dedicated reduction servers would not be used in practice, and was only given as a form of explanation. It was also an important step in researching this scheme, and the basis of the original paper [23].

6

Lookup Table Design

The system model for the Fragmenting Individual Bits (FRIBs) scheme was defined in Chapter 5 but did not describe the lookup operation or how the Lookup Tables (LUTs) obtain fragmented results. This chapter will use a three-server model with the XOR fragment algorithm ($bit = F_A \oplus F_B \oplus F_C$) to further elaborate FRIBs. Simple functions such as addition and multiplication are described, while redundancy and randomisations techniques are introduced. Note that because FRIBs uses a simple lookup to compute over data, there are many more possibilities for what an LUT defines than what is given in this thesis.

6.1 Obfuscating States

With the performance or enhanced privacy model (described in Sections 5.4.1 and 5.4.3 respectively), because each fragment server performs its own reduction, the possible states of the other servers are known. For example, if one fragment server has the state 1011001 ($(1 \oplus 0) \oplus 1$), then the other fragment servers will have the same order of operations; therefore, 1011011 is not a possible state. The result LUT can be thought of as a collection of mini result LUTs, that can be generated for each set of possible states. For this chapter, the result LUT will consist of one order of operations unless otherwise stated.

The representation of Table 6.1 in memory can be a 64-bit number, where the least significant byte is index zero. To perform a lookup, the state is compared against all entries until its index is found. Another way of representing this

Table 6.1: Obfuscated states, mapping to the index.

Index	State
0	1101011
1	11011011
2	10101
3	110101
4	101011
5	1011011
6	101101
7	1101101

Table 6.2: Obfuscated states, mapping to multiple values.

Index	State	Obfuscated States
0	11011011	14, 22, 0
1	1101101	20, 7, 3
2	1101011	9, 21, 8
3	110101	13, 16, 15
4	1011011	1, 23, 5
5	101101	11, 17, 18
6	101011	6, 10, 19
7	10101	4, 12, 2

LUT is by keeping the order of the states as per Table 6.2 (but with one obfuscated state); therefore, the LUT can be a 24-bit number, where the least significant three bits are index zero. Each index would contain a value between zero and seven (which is the obfuscated state), but a challenge is knowing the index of the state quickly. If all the operations are removed and 11 is converted back to 0, then $11011011_2 \rightarrow< 11, 11, 11 >\rightarrow 000_2 \equiv 0$, or $1011011_2 \rightarrow< 1, 11, 11 >\rightarrow 100_2 \equiv 4$. This saves on the amount of memory and storage required for all the mini result LUTs. Table 6.2 can be represented the same way, except 3×5-bits are required per index, in this case requiring 120 bits. With one obfuscated state per fragment state, four operations require 32×5 bits, or 20 bytes. Increasing the number of operations to eight requires $2^9 \times 9$ bits. The number of mini result LUTs depends on the number of different state forms that can be reached. For example, if only addition and multiplication operations are required, many state forms (order of operations) will not be reached.

For the performance model, a different set of result LUTs is used for each fragment server; therefore, with three servers, two obfuscated states are obtained and sent to their corresponding fragment server. For example, fragment server A would obfuscate its state using an obfuscation LUT for fragment server B (the result is sent to fragment server B), and obfuscate its state again but with an obfuscation LUT for fragment server C.[1] When a fragment

[1] An obfuscation LUT is the random mapping between real state and obfuscated state.

server receives the two other obfuscated states, it can get the index into its result LUT. Once again, mini result LUTs can be used because the state form is known–keeping key sizes small. The result LUT is actually a 3-dimensional array where each state is part of the key. If a fragment server's own state is s_0 and the obfuscated states are s_1 and s_2, then the result can be accessed by $s_0 \times n^2 + s_1 \times n + s_2$, where n is the number of possible states. The result value is the reduced state (a single bit) for that fragment server; therefore, with eight states, the result LUT is 64 bytes, where 64 states would require 32,768 bytes. To add randomness, if there are three obfuscated states per state, the result index becomes $s_0 \times 3n^2 + s_1 \times 3n + s_2$.[2]

6.2 NAND Logic

The simplest result LUT configuration is to make the results from NAND logic gates–the proposed operation in the original paper on FRIBs[23]; therefore, arbitrary operations are supported. The result LUTs are formed from the output of the different state (concatenated fragments) combinations. For example, the result of three fragment states $F_A = 101101$, $F_B = 10101$ and $F_C = 110101$ is $(0 \barwedge 0) \barwedge 1 = 0$; this is given below as well to show how the columns form the result. The output for fragment server A could be $result_{lutA}[5, 2, 7] = 0$, where 5 is its state, and 2 and 7 are the two other obfuscated states.[3] Fragment server B could be $result_{lutB}[7, 4, 4] = 1$ and fragment server C could be $result_{lutC}[3, 1, 0] = 1$. Therefore, the resulting fragment is 0, but the fragment servers do not know the result. This will be explained further in this section.

$$
\begin{array}{lccccccc}
F_A= & 1 & 0 & 11 & 0 & 1 & \\
 & \oplus & & \oplus & & \oplus & \\
F_B= & 1 & 0 & 1 & 0 & 1 & \\
 & \oplus & & \oplus & & \oplus & \\
F_C= & 11 & 0 & 1 & 0 & 1 & \\
\hline
 & (0 & \barwedge & 0) & \barwedge & 1 & = 0
\end{array}
$$

[2]Note that the enhanced privacy model only needs one obfuscated state per fragment state because the obfuscated states from the other servers are not known.

[3]The notation $result_{lutX}[s_0, s_1, s_2]$ represents the lookup using three keys.

Table 6.3: Random mappings for fragment server A's result LUT

State	F_A	F_B	F_C
11011	2	2	2
1101	1	0	3
1011	3	1	0
101	0	3	1

Table 6.4: Random mappings for fragment server B's result LUT

State	F_B	F_C	F_A
11011	1	2	0
1101	3	1	3
1011	0	3	1
101	2	0	2

Table 6.5: Random mappings for fragment server C's result LUT

State	F_C	F_A	F_B
11011	1	3	2
1101	3	2	3
1011	2	0	1
101	0	1	0

6.2.1 Sample Tables

An example will now be given to further explain the concept of building LUTs that only reveal the fragment server's state. Only two states will be allowed to be concatenated (single operation) in order to have easier examples, and will use the enhanced privacy model. The performance model can use the same technique for constructing LUTs, but the enhanced privacy model requires more steps, hence it will be used here. Also, the performance model would need many obfuscated states per state for an example this simplistic, where the enhanced privacy model does not. All LUTs are generated on the client and sent to the fragment servers hosted in the cloud. The LUTs can be reused, so this operation only needs to be done on first setup, but it is recommended to change them when possible to reduce any patterns being observed.

A fragment server will obfuscate its state value when sending it to another fragment server; therefore, each requires an obfuscation LUT to be generated with randomised order of states. For example, in Table 6.3, if fragment server C is in state 11011, it will use the value 2 when sending/handling requests for fragment server A; similarly, fragment server B will use the value 1 when it is in state 1011; therefore, Table 6.3 represents the keys into the result LUT for fragment server A. Examples of random mappings for the result LUTs for fragment server B and fragment server C are given in Tables 6.4 and 6.5 respectively. Each fragment server receives (from the client) their column from Tables 6.3, 6.4 and 6.5, meaning fragment server A uses $[2, 1, 3, 0]$ for its own result LUT, $[0, 3, 1, 2]$ for the result LUT used by fragment server B, and $[3, 2, 0, 1]$ for fragment server C, thus giving the obfuscation LUT in Table 6.6. The fragment servers have no knowledge of the mappings on the other

Table 6.6: Obfuscation LUT sent to fragment server A

State	For Itself ($\mathbf{F_A}$)	Sending to $\mathbf{F_B}$	Sending to $\mathbf{F_C}$
11011	2	0	3
1101	1	3	2
1011	3	1	0
101	0	2	1

Table 6.7: Step towards the obfuscated result LUT for fragment server A

$\mathbf{F_A}$	$\mathbf{F_B}$	$\mathbf{F_C}$	Label
0 (101)	0 (1101)	0 (1011)	1010110101011
0 (101)	0 (1101)	1 (101)	101011010101
0 (101)	0 (1101)	2 (11011)	10101101011011
0 (101)	0 (1101)	3 (1101)	101011010101
...	...	...	...
3 (1011)	3 (101)	0 (1011)	1011010101011
3 (1011)	3 (101)	1 (101)	101101010101
3 (1011)	3 (101)	2 (11011)	10110101011011
3 (1011)	3 (101)	3 (1101)	1011010101101

fragment servers. Note that with the enhanced privacy model, the need for obfuscating the state when the fragment server is sending the index and whether the bit should be flipped is not actually required, but to aid explanation and implementation, all inputs will be obfuscated.

The three result LUTs now need to be constructed. An initial step is shown in Table 6.7 for fragment server A, where the obfuscated states $(0, 1, 2, 3)$ from each fragment server are the keys, and a temporary label is used keep track of the real input. This table also has the real values next to the obfuscated states to help link back to the obfuscated mappings given in Table 6.3. Table 6.8 is given as another example for fragment server C, and shows the labels are still in the order F_A 0 F_B 0 F_C. These labels are used to show the link between the three result LUTs, as the result needs to be fragmented across the three tables; thus, all three tables need to be calculated at once.

An example of a possible result LUT for fragment server A is given in Table 6.9; the result bit is fragmented, such that each label is replaced with a bit, where $R_A \oplus R_B \oplus R_C = Result$. This table can be represented in memory as an integer or bit string 1001...1011; in this case 64 bits. Increasing the num-

Table 6.8: Step towards the obfuscated result LUT for fragment server C

$\mathbf{F_C}$	$\mathbf{F_A}$	$\mathbf{F_B}$	**Label**
0 (101)	0 (1011)	0 (101)	1010101011011
0 (101)	0 (1011)	1 (1011)	10101011011011
0 (101)	0 (1011)	2 (11011)	101011011011011
0 (101)	0 (1011)	3 (1101)	10101101011011
...	...	...	...
3 (1101)	3 (11011)	0 (101)	11011010101101
3 (1101)	3 (11011)	1 (1011)	110110101101101
3 (1101)	3 (11011)	2 (11011)	1101101101101101
3 (1101)	3 (11011)	3 (1101)	110110110101101

Table 6.9: A sample result LUT for fragment server A

$\mathbf{F_A}$	$\mathbf{F_B}$	$\mathbf{F_C}$	$\mathbf{R_A}$
0	0	0	1
0	0	1	0
0	0	2	0
0	0	3	1
...	...	...	...
3	3	0	1
3	3	1	0
3	3	2	1
3	3	3	1

ber of concatenated fragments to four with a fixed order of operations would need 4096 bits, and eight concatenations require 2MBs for the result LUT. With the enhanced privacy model sending a smaller LUT across the network, if four rows are requested, then each reduction sends 64 bits and 1024 bits for four and eight concatenations respectively. At 16KBs, as would be the case with 12 concatenated fragments, even though it is still within the maximum Transmission Control Protocol (TCP) size, the Maximum Transmission Unit (MTU) will probably be smaller, meaning the small result LUT will be split across packets; however, as long as the data is streamed, then the RTT will still account for the majority of the processing time.

6.2.2 Using the LUTs

After the previous subsection, the LUTs have been generated for each fragment server: the result LUT that returns the fragment servers new state as a single bit, and the LUT that obfuscates the fragment servers state when sending to the other servers. From the point-of-view of fragment server A, recall that it gets column F_A from Tables 6.3, 6.4 and 6.5 for obfuscating its state, giving Table 6.6. However, its result LUT is actually stored on fragment server B, while it receives the result LUT of fragment server C. Note that the fragment servers receive the LUTs from the client or user, so they know nothing about their own result LUT, or any of the other tables that it was not sent.

For a reduction–which in this example will need to happen each operation over fragments–fragment server A will first obfuscate its state according to Table 6.6, giving the obfuscated states O_A, O_B and O_C. It then generates a random vector to request rows from its result LUT stored on fragment server B. For example, the vector could be $< 1, 0, 3, 2 >$, where the index of O_A is the actual row it will use to get its new state. With a larger number of possible states, this vector will be a subset of the result LUT, but in this case, all rows can be requested. The vector is sent to fragment server B, while fragment server C will send a similar vector to fragment server A, because fragment server A has the result LUT for fragment server C. Upon receiving this vector from fragment server C, the result LUT is first reordered; for example, with the vector $< 1, 0, 3, 2 >$ and a result LUT of 64 bits, the order will become $bits_{8-15}$ $bits_{0-7}$ $bits_{24-31}$ $bits_{16-23}$. The reordered LUT (and in cases where a subset is requested, smaller reordered LUT) then needs to be obfuscated. This is done through pseudo-random number generators, where a 64-bit number can be generated and XORed with the reordered result LUT. More values are generated to shift the groups of bits, where each set of bits is rotated using different values (note that the shift could be replaced with a shuffle function). This obfuscated result LUT can then be sent back to fragment server C. In parallel (at the same time), fragment server B has sent fragment server A an index value into the randomised vector it sent fragment server C when it was requesting its result LUT, using the same seed (for the pseudo-random number generator) that fragment server C will use for obfuscating the result LUT that fragment server B will receive. Fragment server A can use its

obfuscated state for fragment server B to get the new index–since the result LUT is shifted/rotated–and see if that bit will need to be flipped. The index and flip value are sent back to fragment server C. To summarise, a fragment server is computing an obfuscated result LUT, and an index and flip value in parallel.

Fragment server A receives its obfuscated result LUT from fragment server B, and an index plus flip value from fragment server C. It knows the index of the group of 8 bits in the received obfuscated result LUT from its own state (the index into the vector sent to fragment server B), then uses the index from fragment server C to get the bit from the group of 8 bits. Finally, it will flip the bit if needed before the final result fragment is revealed. This section has been similar to the example in Section 5.5.1, but has given a detailed look into the use of LUTs and how they are created using NAND function.

6.2.3 Network Data Transferred

To summarise the data being transferred across the network during a reduction request, this section will briefly cover the scenario of six concatenated fragments. The performance model has only a few bits transferred for each reduction request. With 64 reachable fragment states for each order of operations, only 6 bits are sent between each of the fragment servers (excluding packet data). With 100 fragments being reduced at once, 700 bits are sent between each fragment server. Labels or ordering information are not required, because each fragment server is running the same instruction set; therefore, order is already known. With TCP connections already established, only half the latency of RTT is experienced.

As mentioned in the previous section, the enhanced privacy model sends a small obfuscated LUT, an index vector, an index and flip bit between each fragment server; the last three variables do not add much data to be transferred. With 64 states, a vector of length 4, an index into that vector and flip bit would need 31 bits–already more than the performance model. The small obfuscated LUT is much larger at 32 bytes, giving approximately 36 bytes per fragment. Therefore, for 100 reduction requests, 3.5KBs would be transferred between each fragment server. This is far less than given in Table 3.3 for an existing multi-party computation scheme, where only tallying 6 votes required

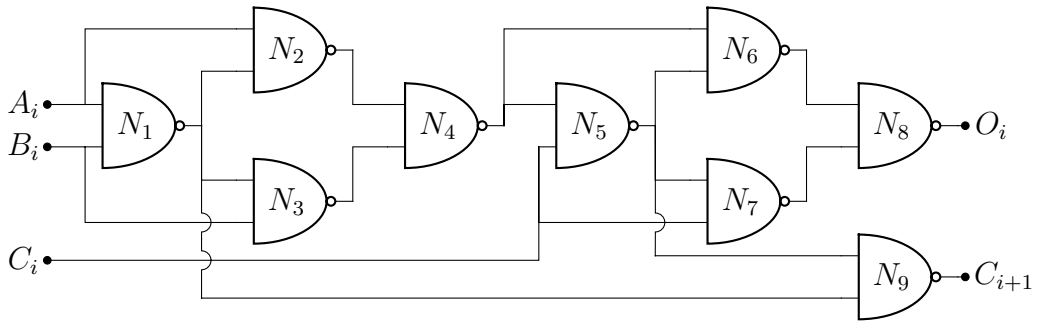

Figure 6.1: NAND gate full adder

approximately 10KB. Even with the data split over multiple packets, when the first packet is received, processing can occur straight away, meaning once the reduction requests are processed, the next packet should have already arrived. So long as each small obfuscated LUT can fit in a packet, then with multiple reduction requests, if data is split over multiple packets, it should not affect performance in any significant manner.

6.3 Simple Operations

In this section, the NAND result LUTs described in the previous section will be used to define two simple operations: addition and multiplication.

6.3.1 Addition

The addition of two 32-bit integers can be achieved with 31 full-adders and a single half-adder. A full-adder comprised of NAND gates can be seen in Figure 6.1. 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 in Figure 6.1–which for worst-case is where A_i and B_i are both 0–giving $(A_i \overline{\wedge} (A_i \overline{\wedge} B_i)) \overline{\wedge} (B_i \overline{\wedge} (A_i \overline{\wedge} B_i)) \rightarrow 110110110011011011$. The fragment therefore can grow up to 18 bits during this step, or more importantly 2^6 possible states. We can then combine all 32 fragments for N_4 into a single network payload and send them to be reduced to single bits. Therefore, all N_4 states become a

single bit for the cost of one reduction[4]. With each fragment server executing the same program, the order of states will be consistent; therefore, no labels are required.

The method for how the carry bits are reduced can vary depending on implementation, as we need to define how large the fragments are allowed to grow. A smaller size requires more reduction requests, thus decreasing performance. Because the first bit does not have an input carry value, the input for N_9 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 reduced fragment, and that the worst-case value for N_1 is 11011, each carry step will at most add 8 bits to the fragment (11011011) or 2^3 possible states multiplied by the existing state possibilities. If the enhanced privacy model is used, at the same time all N_4 gates are being reduced, all the N_1 gates could also be reduced. This would mean the N_9 input from N_1 is now a single bit, meaning only a multiple of 2^2 states are added to the LUT. Furthermore, if the result LUT is only used for the addition carry bit, then even if N_1 is not reduced, only 2^2 states are added, because the left-most and centre fragments are equal (101011 is valid, where 1011011 is not). Note that this could also be applied to N_4, where there are only two input states. Ultimately, there many different ways of constructing this carry operation; for example, the carry could just be the input values A_i and B_i, where padding is zeros, such as $00B_2A_2B_1A_1C_1$, where the first carry is a single bit. This also highlights why the enhanced privacy model is critical to FRIBs, because when only a few possible states are reachable, then the performance model knowing the others states in not ideal. In this case, the performance model would require many obfuscated states for a single state; therefore, the enhanced privacy model is superior.

Knowing that the order of operations will be constant, means only the reachable states need to be in the obfuscation LUT (mapping from state to obfuscated value) specifically used for addition carry. The length of the fragment states would vary, so padding is required for the N_9 gates earlier, using known *high* or *low* values, such as, $1 \barwedge 1 \barwedge 1 = 1$ and $0 \barwedge 1 \barwedge 1 = 0$, where $\barwedge 1 \barwedge 1$

[4]Depending on the amount of data being transmitted, if many packets are required, there will be a small amount of time added compared to a single reduction.

provides padding of length 2. Padding of length 2 is important, as that is how many fragments are concatenated for each carry. Once the number of allowed carryovers has occurred, a reduction takes place for all N_9 gates up to that point (with padding), repeated until all N_9 gates have been reduced. Finally, like N_4, all N_8 reductions can be grouped together and sent in the same reduction request, which gives the final output, because all inputs to N_5 and N_7 are now a single bit (a reduced state). The result is still hidden from the fragment servers.

With this approach, the minimum number of reduction requests would be 3: all N_4 gates, all carry bits, then all N_8 gates. However, this produces a very large and impractical LUT for the carry bits (where the LUT for both N_4 and N_8 would remain the same) for a 32-bit number. Dividing the operation up such that four 8-bit values are added together still requires a few terabytes. For small LUT sizes, adding eight 4-bit values together is more optimal, requiring only a few hundred kilobytes. For the enhanced privacy model, this means transferring a small obfuscated LUT that is only a few kilobytes. Therefore, 10 reductions are required in total, meaning that the worst network latency at 10ms (20ms RTT), the cost for the performance and enhanced privacy model would be $\approx$ 100ms and $\approx$ 200ms respectively. If the inputs A_i and B_i are used directly instead of the NAND gates, the number of requests drops to 9, and consists of the carry operations and all output bits (same as N_8). The performance model can actually have an LUT for 6-bits, meaning only 6 reduction requests are required for the carry (without additional obfuscated states), and the enhanced privacy model can increase to 5 bits for the carry, giving 7 reduction requests for the carry. With the performance model, the number of obfuscated states per state will greatly affect the size of the LUT, where with the enhanced privacy model, the small obfuscated LUT sizes need to be considered for network transfers and obfuscation.

6.3.2 Multiplication

Binary multiplication can be thought of as a series of AND operations added together. Below (next page) is an example of multiplying 5 and 11 on an 8-bit machine. For each bit in 11_{10}, we *AND* it with each bit in 5_{10}, giving 8 values; adding each of these values together gives 55.

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

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

$$
\begin{array}{rrrr}
00000101 & 0000 & 000000 & 00 \\
+\ 0000101 & +\ 000 & +\ 00101 & +\ 0 \\
\hline
00001111 & 0000 & 001010 & 00
\end{array}
$$

This step is repeated below.

$$
\begin{array}{rr}
00001111 & 0000 \\
+\ 001010 & +\ 00 \\
\hline
00110111 & 0000
\end{array}
$$

The final addition gives the result:

$$
\begin{array}{rl}
00110111 & \\
+\ 0000 & \\
\hline
00110111 & \therefore = 00110111
\end{array}
$$

This gives a total of seven addition operations for this example, but by adding similar sized numbers together in parallel, we can reduce the number of reduction steps required. When adding multiple values in parallel, 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 slower than three additions. For 32-bit values, there are a total of 31 additions, but they perform like five additions.

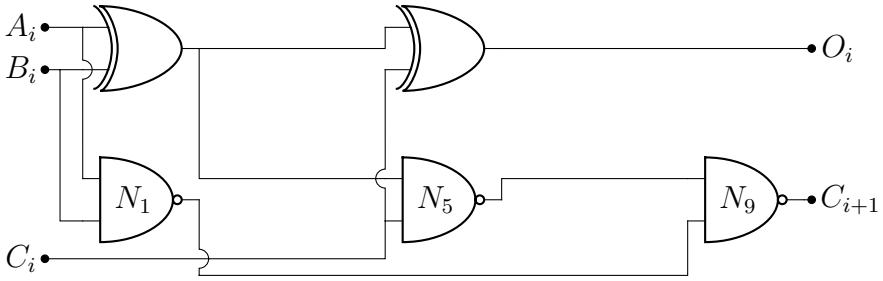

Figure 6.2: NAND and XOR gate full adder

6.3.3 Free Operations

A simple operation that can be free is bit shifting by a constant, if the shift value can be known.[5] Depending on the fragmentation algorithm, another free operation can be available. The fragmentation algorithm used primarily in this thesis is XOR; therefore, the XOR operation can be applied for free on each fragment server without a need for reduction, as shown below. Two sets of fragments are XORed; each fragment server can XOR its two fragments together, where the result fragment produces the correct value.

$$\begin{array}{r} 1 \oplus 1 \oplus 1 = 1 \\ \underline{0 \oplus 1 \oplus 1 = 0} \\ 1 \oplus 0 \oplus 0 = 1 \end{array}$$

The full adder made from NAND gates actually consists of two XOR operations, as shown in Figure 6.2. Therefore, the reduction requests for N_4 and N_8 can be removed and performed for free. Building the carry operation such that 4 bits are concatenated together would now only require 8 reduction requests. Another function that is free with an XOR fragmentation algorithm is NOT functions. If only one fragment server has an instruction to flip its result, then this is equivalent to a NOT function This same concept will be used to provide randomness in Section 6.5.

[5] A free operation is where the operation can be applied without information from the other fragment servers.

Index	State	Obfuscated State
0	00000000	242
1	00000001	51
2	00000010	466
3	00000011	163
...	...	...
508	11111100	135
509	11111101	245
510	11111110	64
511	11111111	159

6.4 Other LUT Operations and Functions

This section expands on generic NAND logic, by providing examples of targeted result LUTs.

6.4.1 Addition

Section 6.3.1 detailed how to perform an addition operation using NAND gates and introduced the idea of specialising LUTs to reduce LUT sizes. This idea will be explored further, where the result is 5 bits, instead of a single bit. An example of 8 bits (4 bits of A, and 4 bits of B) plus a carry is given. The first operation will need to set the carry to zero. The output from the result LUT will be 4 bits plus a carry. The result LUTs are a moderate size at 80MB, but a 32-bit integer is only going to require 8 reduction requests. Each small obfuscated LUT that gets sent would be 320 bytes per row requested. With this type of operation, using the enhanced privacy model is recommended.

6.4.2 Multiplication

When compared to addition, multiplication is more difficult, because the resulting values are twice the size of the input, where the result of 4 bits $\times$ 4 bits is 8 bits. The bits are also not just column based but affect each other column; therefore, no further time was spent on designing a dedicated multiplication LUT, where just using the addition function and *AND* function will be efficient for the purpose of this thesis. A speedup could be realised by building a result LUT for 4-bits multiplied by 4-bits, and adding the 8-bit results to-

gether. Below shows that only four addition operations are required, halving the number previously required. With reduction requests parallelised, this will perform similar to two additions. However, for 32-bit integers, it will perform worse.

$$
\begin{array}{r}
00010101 \\
\times\ 00011011 \\
\hline
00110111 \\
00001011 \\
00000101 \\
+\ 00000001 \\
\hline
0000001000110111
\end{array}
$$

6.4.3 Conditional Statements

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 encrypts 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 either encrypt a 0 or 1 and the other subgroup must encrypt the opposite. However, our proposed scheme has the bits fragmented across many servers, meaning all the servers must compute over the same instruction set. This prevents a compromised server 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.

Secure Results

Returning results securely means the result is a fragmented bit, where none of the fragment servers have 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 4 and 5, 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 4 If equals example

```
1: if a = b then
2:     c ← c + 1
3:
4: function ifEqual(a, b)
5:     m ← a − b
6:     inout ← 0
7:     carry ← 0
8:     for i ← 0 to 32 do
9:         tmp ← m[i] + inout + carry
10:        inout ← tmp & 1
11:        carry ← tmp >> 1
12:    return !(inout | carry)
13: c ← c + (1 × ifEqual(a, b))
```

Algorithm 5 If greater than or equal example

```
1: if a >= b then
2:     c ← c + 1
3:
4: function ifGreaterEqual(a, b)
5:     sign_neq ← a[31]^b[31]
6:     c ← a − b
7:     return (!sign_neq & !c[31]) | (sign_neq & !a[31])
8: c ← c + (1 × ifGreaterEqual(a, b))
```

Non-Secure Results

Instead of returning a fragmented bit, this approach returns the whole bit by using a different set of LUTs 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 faster to compute. However, there is more risk associated with this method, so the secure method should be considered first.

6.4.4 Modulus

The work presented in Appendix A details a custom modulo algorithm developed as initial research into the research question for this thesis. This algorithm uses a simple LUT to compute the modulo function; therefore, it can easily

be used with FRIBs (assuming a static modulo value). Bit shifting is a free operation, and a custom LUT can be created to add the overflow value, since this value is likely to be smaller (for example, adding a single bit to a 32-bit value). A static modulo value could be used to encrypt the data within the fragments, or to prevent array overflows. If the modulo value is dynamic, then the value that is being added with each overflow needs to be calculated first.

For a 32-bit value, modulo result LUTs of 8 bits can be constructed, meaning the best case scenario is only a few shift lookups and additions are required. The challenge with this algorithm is knowing if an overflow occurred after the shift addition is complete, because when this happens, another lookup and addition is required. With the worst-case overflow for addition being a single bit, the modulo LUT can include an extra bit, resulting in 8 bits plus 1 bit to handle the possibility of an overflow occurring previously. Note that this will work with larger numbers as well, such as 2048-bit values. This does not remove initial issues, as the final subtraction step needs to know when to stop subtracting such that the result is correct. Depending on the requirements, having the result a few bits larger could be acceptable. If not, then the program is going to need to know when to stop subtracting. This is a case where the non-secure conditional statement mentioned in the previous section could be used. In terms of privacy, this could reveal how close the modulo algorithm got to producing the right result but it will still difficult to learn meaningful information.

An approach to mitigate this would be to have another lookup for subtraction, where a set number of subtractions occur before checking if the result is less than the modulo value. This subtraction would either remove modulo or zero from the result. Also, because the algorithm specifies working in blocks of size of the modulo value, this could help reveal the highest order bit of the modulo value. Therefore, the block size should be greater, but this then requires more subtractions at the last step. Ultimately, this is a challenging algorithm to implement in FRIBs, but it is possible.

6.4.5 Hidden Operations

With variable operations, the ability to try and hide the operation is possible. Instead of the program saying *ADD*, this could be replaced with *FUN1*. The

fragment servers would know to use the LUTs associated with *FUN1*; however, it may be possible to guess the function based on how it is used. For example, there is a big difference between an addition and multiplication operation because of the number of reductions required. Hence, hiding the program is out-of-scope of this thesis but worth mentioning.

6.5 Randomisation

The randomisation properties of an encryption algorithm affect the overall strength of the scheme. The same is true with any obfuscated or hidden value such as the fragments in FRIBs; therefore, this section will demonstrate how to add randomness during processing.

6.5.1 Randomising Result Values

With the enhanced privacy model and three fragment servers, a single fragment server has control over whether two of the three resulting fragment bits are flipped or not. Given the three values in the equation $a \oplus b \oplus c$, flipping any two bits does not change the result; for example, $1 \oplus 1 \oplus 0 = 0$, flipping two bits gives $0 \oplus 0 \oplus 0 = 0$, $0 \oplus 1 \oplus 1 = 0$ or $1 \oplus 0 \oplus 1 = 0$. Therefore, to add some form of randomness to the results, a fragment server can choose to flip two bits. It does this by flipping its own resulting bit, and by toggling the flip value[6] sent to another fragment server.

A slower technique, but one that will support the performance model, is where one fragment server is chosen at random to be the randomiser. The other two fragment servers send the result fragment encrypted using the Goldwasser–Micali cryptosystem. Each fragment server uses its own key to encrypt its fragment result so only it can see it. When the randomiser fragment server receives the encrypted results, it flips an even number of bits: either none or two. This keeps the overall result the same but changes the result fragment. However, this is much slower and less efficient than the previous method.

[6]The value sent along with the index into the small result LUT.

6.5.2 Randomising Fragment Values

The same idea (for randomising result values) can be used to randomise the stored fragments. Using a shared seed for a pseudo-random number generator, each fragment server can decide whether to flip its fragment or not. For example, we can generate a number r between 0 and 100, where most values do not change operations, but if r_{AB} (such as 56) is generated, fragment servers A and B flip their fragment value. This would allow all stored fragments to be randomised periodically.

6.6 Redundancy and Malicious Fragment Servers

Processing data–especially sensitive information or mission critical information like voting–needs some form of redundancy, while the ability to handle purposeful corruption is also desirable. This would prevent a malicious or compromised server from corrupting the results.

6.6.1 Parity Bit

Using a single parity bit for the fragments when bit fragmentation is achieved with XOR gates means the parity bit may not reveal any information, whereas if the fragmentation algorithm used all *AND* gates, the parity bit would reveal the value. However, using three fragment servers in order to support the presented enhanced privacy model would mean only two fragments are inputs into the fragmentation algorithm, and the remaining fragment is for parity. Therefore, the parity bit reveals the value, where if $F_A = 0$ and $F_B = 1$, the parity bit must be 1 (assuming *high* bits modulo 2 is zero for parity). Even with the assumption of only one server containing corrupt data, data cannot be recovered during processing; however, the user could be notified of the corruption and processing can be halted. The user can then try and solve which server is corrupting the data in order to attempt to recover it.

Allowing for more fragment servers would mean multiple parity bits could be used, stopping an individual parity bit representing the plain-text value. An example is shown in Table 6.10 and 6.11, where two bits are processed using the XOR fragmentation, such that $bit = F_A \oplus F_B \oplus F_C$ and parity

Table 6.10: Parity bit examples with three initial fragments and two parity bits

Bit	$\mathbf{F_A}$	$\mathbf{F_B}$	$\mathbf{F_C}$	$\mathbf{p_0}$	$\mathbf{p_1}$
0 =	1	0(11)	1	0(11)	0(11)
1 =	1	1	1	1	0(11)

Table 6.11: Parity bit example showing a corrupt fragment

Fragment	Valid	Corrupt
F_A	101	101
F_B	1101	11011_{err}
F_C	101	101
p_0	1101	1101
p_1	11011	11011
Result	1101	110??

bits p_0 and p_1 check the number of *high* bits is zero modulo 2. Table 6.10 shows the two sets of fragments, each with three data fragments and two parity fragments. Combining these two sets of fragments together gives the left column of Table 6.11. The result of the left column is 1101 and importantly the parity bits show that both sides of $\{F_A, F_B, F_C, p_0, p_1\}$ have an equal number of high bits. However, in the right column, F_B has been corrupted[7], which gives a parity error for the rightmost side of $\{F_A, F_B, F_C, p_0, p_1\}$; therefore, the result value is unknown and the user can be notified. Ultimately, because parity bits do not allow correction during processing and only one fragment server can be malicious, it is not beneficial to privacy-preserving processing.

6.6.2 Linear Block Codes

Another method of error checking is by using binary linear block codes, which, unlike parity bits, can support data recovery during processing. This increases the number of fragments required; for example, three fragments become five fragments (for redundancy detection) or six fragments (for automatic recovery); however, both only require three fragments for decoding. This section will focus on six fragments, as it will be used for a voting implementation in Section 7.2. Redundancy is achieved by trying all the combinations of three out of the six fragments. If all the decoded values are the same, then no cor-

[7]The x_{err} notation indicates an error or corruption to the fragment.

Table 6.12: Erasure encoding for three sets of fragments

Value	Fragments	Encoded
1	111	111000
0	101	101101
0	011	011110

ruption has occurred. However, if one fragment is corrupt then only decoded values which included that fragment will vary. Recovering automatically from multiple corrupt fragments becomes similar to the parity bits method, where the user input is required.

$$G = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

With the fragmentation algorithm $F_0 \oplus F_1 \oplus F_2$, Table 6.12 shows some values being encoded with FRIBs, then with the generator matrix G shown above. For example, the fragment value 101 becomes 101101 by vector multiplication within modulo 2. The second bit in 101101 is calculated by $[1\ \ 0\ \ 1] \times [0\ \ 1\ \ 0] \rightarrow 0 \oplus 0 \oplus 0 = 0$.

$$[1\ \ 0\ \ 1] \times \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} = [1\ \ 0\ \ 1\ \ 1\ \ 0\ \ 1]$$

These three encoded fragment sets are combined, giving the middle column of Table 6.13. The right column contains the same values, but F_A has one corrupt value. The valid concatenated value for F_A is the leftmost bits from the encoded column in Table 6.12: 1, 1 and 0 giving 101011.

Table 6.13: Erasure encoding example after three concatenations

Server	Valid	Corrupt
F_A	101011	1011011_{err}
F_B	101101	101101
F_C	10101	10101
F_D	110101	110101
F_E	1101101	1101101
F_F	1101011	1101011

Using F_C, F_D and F_E from the corrupt column of Table 6.13, the values 100, 110, and 111 are obtained. The value 100 is the leftmost fragment (1, 11, and 11) of the corrupt column for rows F_C, F_D and F_E. Decoding with generator G, then with the fragmentation algorithm, gives 1, 0 and 0, which are the original bits (plain-text) in Table 6.12, shown below. Note that decoding from generator G can be achieved by finding which inputs for F_C, F_D and F_E give the encoded values ??100?, ??110?, and ??111?.

??100?	??110?	??111?
$\Downarrow$	$\Downarrow$	$\Downarrow$
G	G	G
$\Downarrow$	$\Downarrow$	$\Downarrow$
111	101	011
$\Downarrow$	$\Downarrow$	$\Downarrow$
$F_0 \oplus F_1 \oplus F_2$	$F_0 \oplus F_1 \oplus F_2$	$F_0 \oplus F_1 \oplus F_2$
$\Downarrow$	$\Downarrow$	$\Downarrow$
1	0	0

Using F_B, F_D and F_E gives the correct values as well, as shown below.

?1?00?	?0?10?	?1?11?
$\Downarrow$	$\Downarrow$	$\Downarrow$
G	G	G
$\Downarrow$	$\Downarrow$	$\Downarrow$
111	101	011
$\Downarrow$	$\Downarrow$	$\Downarrow$
$F_0 \oplus F_1 \oplus F_2$	$F_0 \oplus F_1 \oplus F_2$	$F_0 \oplus F_1 \oplus F_2$
$\Downarrow$	$\Downarrow$	$\Downarrow$
1	0	0

However, using F_A, F_D and F_E below gives an incorrect value, because of the corruption to F_A.

1??00?	$0??10?_{err}$	0??11?
$\Downarrow$	$\Downarrow$	$\Downarrow$
G	G	G
$\Downarrow$	$\Downarrow$	$\Downarrow$
111	010_{err}	011
$\Downarrow$	$\Downarrow$	$\Downarrow$
$F_0 \oplus F_1 \oplus F_2$	$F_0 \oplus F_1 \oplus F_2$	$F_0 \oplus F_1 \oplus F_2$
$\Downarrow$	$\Downarrow$	$\Downarrow$
1	1_{err}	0

By trying all the combinations (${}^3C_6$), the results are only wrong (or different) when F_A is used. This allows processing to continue even with a fragment server compromised or producing corrupt values.

With the performance model, implementing linear block codes is straight forward, as a large result LUT with six keys/inputs can be constructed. This is because the block code (the six bits) can be treated the same as a fragment algorithm when constructing the LUT offline. However, the enhanced privacy model currently only supports three keys/inputs. With only three inputs required to decode the 6-bit linear block code, the same concept can be used, but with multiple result LUTs. Using six bits, means requiring six fragment servers, where each combination of three bits is tried, resulting in twenty sets of LUTs required for each fragment server. If the fragment server assumes it is not corrupting data (because if it were then it would corrupt the output as well), this reduces the number of sets of LUTs to ten. The negative of this assumption is that if an accidental corruption occurred, it would not be fixed straight away[8]; however, it does mean that purposeful or malicious corruption on a single fragment server cannot corrupt the overall result.

The result for the fragment server is the majority result from the ten result

[8]The corrupt fragment may be fixed as more processing occurs.

LUTs. This does mean that the result fragments need to be the same across all ten result LUTs (which can be handled offline when generated), such that with a non-corrupt system all the fragment servers' result LUTs give the same fragmented bit. The other consequence of this is that fragment servers cannot randomly flip bits; however, this is not really necessary, with the protection that the enhanced privacy model gives. To expand further, we build the LUTs with only three inputs, and compare the outputs of each, where we select the output ($\in \{0, 1\}$) that occurs more frequent. Hence, the result of each result LUT needs to be the same.

The RTT is still the major factor for computation time, where the largest RTT between any of the six fragment servers (because there are now six bits) can still be used to estimate overall processing times. Depending of the size of the LUTs, performance may be slightly impeded by more data needing to be in memory, and more frequent cache misses. Also, by each fragment server needing to do more lookups and obfuscating, it will also negatively impact performance. Therefore, supporting the ability to recover from errors or a malicious fragment server will come at a cost of performance, but RTT and bandwidth are still the main overheads.

In Chapter 7, the enhanced privacy model is analysed with and without redundancy. The technique used to allow the enhanced privacy model to support six fragment servers is to construct the result LUTs using six keys. Therefore, if one key was corrupt, the result would still be the correct state. The challenge was keeping the network traffic to a minimum for greater performance. To accomplish this, a fragment server will send a vector to every server, as shown in Figure 6.3 . This means when the fragment server is generating the small obfuscated result LUT, it need only contain 4^5 entries (four values in the vectors). The index into each vector also needs to be sent to the fragment server handling the index and bit-flip calculation as well; therefore, in total each fragment server is receiving five vectors, and five indexes.

Figure 6.3 shows a similar process to Figure 5.14. They start to differ at step 3, as now six obfuscated states are obtained from the fragment servers' original state. Fragment server A then generates vectors for each of the other fragment servers, where the vector for D is actually sent to fragment server E. There is no vector for fragment server F, as fragment server A is responsible

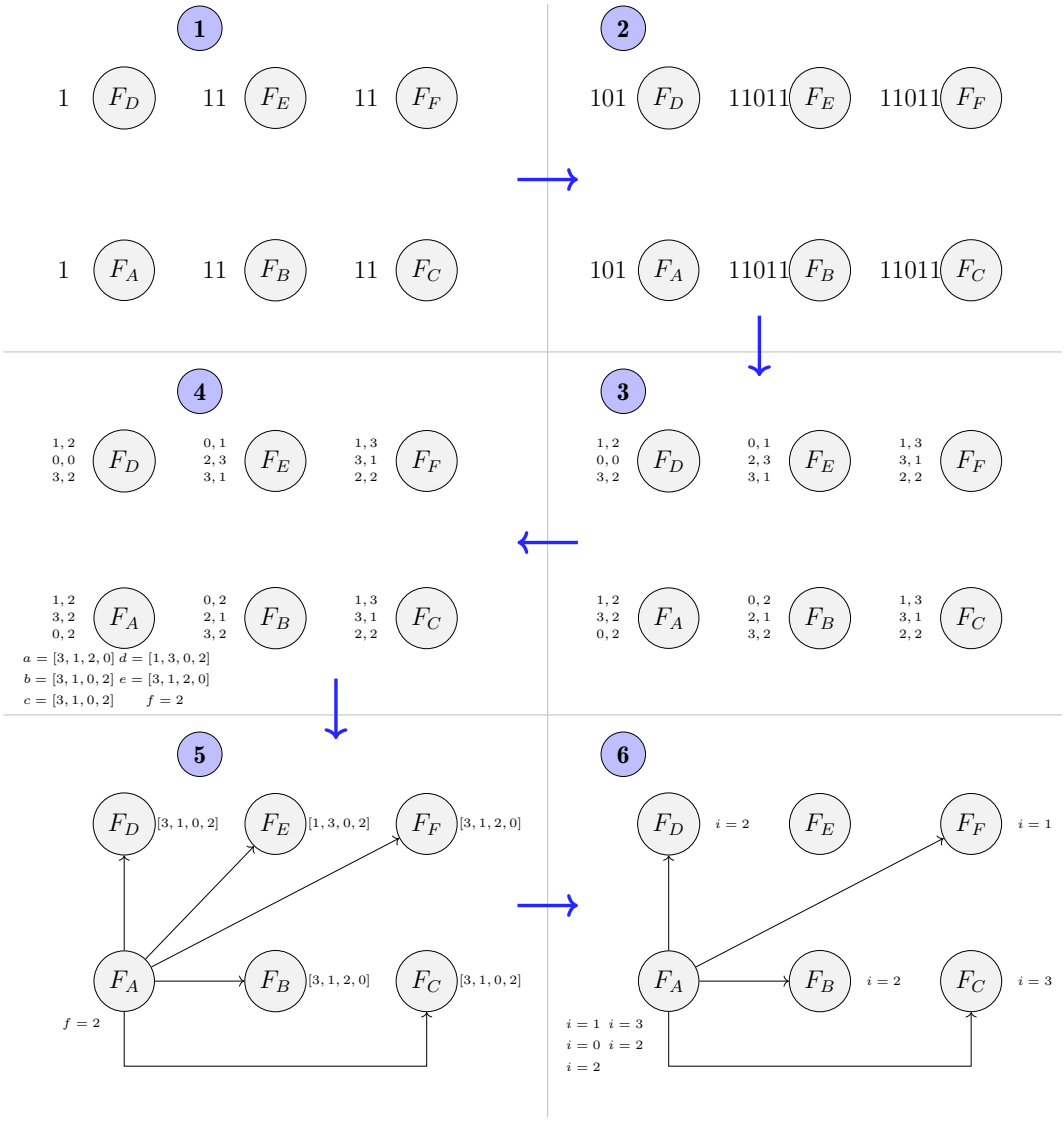

Figure 6.3: Reduction flow for fragment server A with built-in redundancy support

for generating the small obfuscated LUT for fragment server F. Once step 5 has sent the vectors to all other servers, step 6 sends the indexes for those vectors. For example, with the vector sent to fragment server B, the index of the correct value is sent to fragment server F. No value is sent to fragment server E, and fragment server A calculates the index and flip value for fragment server F. The results in Chapter 7 for the redundancy model show useable performance, even with the built-in redundancy and the fact there are six keys for the result LUTs.

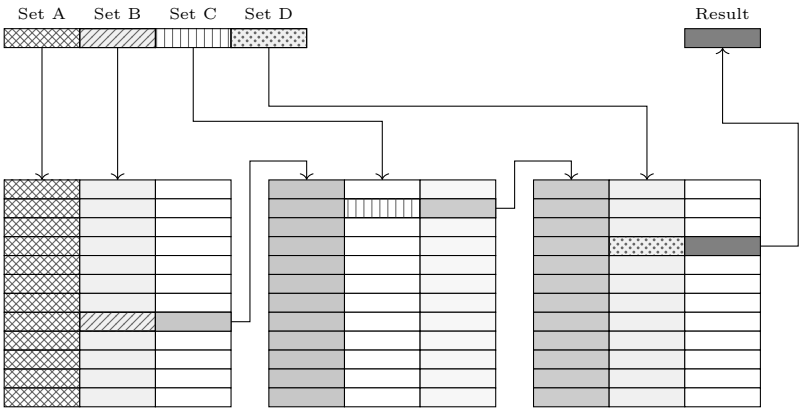

Figure 6.4: Linked LUT flow diagram

6.6.3 Corrupting Fragments

In Section 5.4.3, an approach is used such that no fragment server can learn the states or repeating of states when a reduction occurs. This can be improved further by purposely corrupting fragments sent to the servers from the client. With the performance model, if redundancy is not required, every bit can have a corrupt fragment. Therefore, a fragment server does not know which one the corrupt fragment is, making it harder to learn anything during the reduction stage. Furthermore, if a fragment server receives two obfuscated fragments with the same value, they are not guaranteed to be equal.

6.7 Linking Tables

With network latency affecting performance, reducing computation time means either smaller RTTs, or fewer reduction requests. The RTTs will vary based on the location of the fragment servers; however, limiting reduction requests means more fragments must be combined before a reduction occurs. With three fragment servers or three inputs as in the case of redundancy, six concatenated fragments will produce a 32KB result LUT for one order of operations. Increasing to 12 concatenated states, halving the number of reductions required, it now has a 8GB result LUT. For the enhanced privacy model, where part of the result LUT is transferred over the network, this is not ideal.

With six concatenated fragments, there are $(2^6)^3$ possible states or results.

If one more concatenation is allowed, the possibilities becomes $(2^6)^3 2^3$ or $(2^7)^3$. However, splitting the concatenations into groups of four and linking the result LUTs together will still give the same result but needs less space/memory. For example, the first result LUT $s = (2^4)^3$ giving $slog_2(s/2)$ as the result is no longer a single bit, and requires 1MB. Now, combining with another LUT, where the first set of four concatenated fragments are combined with another set of four concatenated fragments, gives $(((2^4)^3)/2)((2^4)^3)$ possibilities. Keeping the output states to 11 bits gives 11MB. Then, the next set of four concatenated fragments can be combined in the same manner, requiring another 1MB LUT, which has the result for 12 concatenated states. Figure 6.4 shows an example of four sets of concatenated fragments being linked together. This totals to $\approx$ 13MB for the result LUTs, down from 8GB.

For the performance model, this is easy to accomplish, as the fragment server holds its own result LUT. However, the security or strength is weakened due to two outputs giving the same obfuscated state to be used or linked to the next LUT, meaning both must be the same result bit after x concatenations. The enhanced privacy model is more difficult, because each fragment server does not store its result LUT, and privacy is achieved by the result only being a single bit. Therefore, this concept was not explored further.

6.8 Summary

This section has shown that most functions can be implemented by using precomputed LUTs, and with the enhanced privacy model, the LUTs can be reused. Generation of the obfuscation and result LUTs is simple, and once computed on the client can be uploaded to the fragment servers. The enhanced privacy model can reuse these LUTs, because only a single bit is exposed, whereas the performance model will require frequent generation of new LUTs to prevent patterns being observed. However, as Section 6.2 shows, generation requires little effort and could even be computed on a mobile device. As the modulo function in Section 6.4.4 proves, some algorithms or functions that need the conditional operation can be more problematic but are still possible.

7

Experiments and Analysis

This chapter presents proof-of-concept implementations for FRIBs for two applications: (1) secure cloud voting, and (2) secure string searching. The first is a dedicated implementation written in Python, which aimed at providing useable performance, while keeping votes hidden. Secure searching is a flexible implementation in Lisp to show that FRIBs is practical in areas other than performance. The data privacy strength of FRIBs is analysed and shows that with an even distribution of *high* and *low* bits, no state bias is experienced. Finally, this chapter introduces the idea that distributing data has benefits for companies and personal users in regards to data privacy laws, making it more difficult to retrieve all the fragments.

7.1 Prerequisites

Before the practicality of FRIBs can be evaluated, there are two pre-processing steps that need to be explained: (1) how to keep the fragments in order while processing, and (2) generating the LUTs for FRIBs to use.

7.1.1 Fragment Scheduler

When an organisation or entity is processing their own data in the cloud, such that they manage all fragment servers, then scheduling may not be required, because they know where and how the data is stored. This means if the program needs some value x, then the fragments can be passed as an argument to the program on each fragment server. However, when data is sent to the

fragment servers from a client and processed automatically, then the order of the fragments needs to be guaranteed. This is because if multiple clients are submitting data at the same time, the fragment servers could receive the fragments in a different order to each other, causing corruption during processing. Adding identifiers to each fragment during reduction would be one method of ordering the fragments; however, with the voting use case where a single tally is being incremented, each addition references the previous operation, requiring the order to be defined before any operation occurs. This section will briefly introduce a simple fragment scheduler to address this problem.

Each fragment server is responsible for a pool of fragments, where each pool is processed one after another, similar to a circular buffer. Each fragment server has the same set of pools, but each pool has a controlling fragment server to define order. When a fragment received is for a pool managed by that fragment server, it adds the fragment to the pool and records the index in the pool. The fragment server then sends the voter identification number (or another form of identifier) and index to the other fragment servers. Upon receiving this tuple, the other fragment servers can add the fragments for that voter (or identifier) into the pool at the same index. This guarantees the ordering in the pools, as only one fragment server is deciding the position. Note that the fragments are currently distributed between the pools by the voter identification number within modulo number of fragment serves.

To add support for a corrupted or malicious fragment server, when the next pool for processing is being staged, the fragment servers first check the number of fragments, and then that the ordering of each pool is the same. This is achieved by each sending a signature of the pool, by hashing the voter identification numbers in order, and sending the signature to the other fragment servers. If all the signatures are the same, the pool is processed, otherwise the pool needs to be rescheduled. The rescheduling will be done by another fragment server (which can be the previous pools fragment server); meanwhile, the next pool will be used.

The flow for how the scheduler gets the next pool of fragments is given in Figure 7.1. This can be seen as a state machine for a subthread of the scheduler to handle rotating the pools. A short description of each state is given below.

- **Init:** The entry point for the thread.

- **Conn:** Connect to all the other fragment server schedulers.
- **Next:** Start the process of getting the next fragment pool. There could be a delay or a trigger that signifies that the next pool is required.
- **Len:** Send the length of the next pool queue to the other fragment servers.
- **Recv$_0$:** Receive all the lengths of the other fragments servers. Technically it only needs the length from the fragment server managing that pool; however, they can be used to detect malicious activity.
- **RLen:** Wait until the length of the next pool queue is the same length as the managing fragment server. This is in case there has been a delay in receiving an index and guarantees the lengths of the next pool on each fragment server is the same. Any difference in length would cause corruption for all following reduction requests. A timeout would need to be added here.
- **Sign:** Send the signature of the next pool to the other fragment servers. This can be accomplished in many ways, where for proof-of-concept the fragment identifiers are hashed to guarantee the order of the next pool. Any variation in the ordering would again cause corruption. For the voting example given in this chapter, the voter identification number can be used as the fragment identification.
- **Recv$_1$:** Received all the signatures from the other fragment servers. This allows the detection of malicious activity, as all the signatures should be equal.
- **Q:** This state is reached if the signatures are equal, or in the case of the redundancy model, where all but one signatures are the same. The state adds the fragments in the next pool to the processing queue.
- **Fix:** A difference in the signatures occurred, so the fragments cannot be added to the processing queue. Another fragment server will attempt to sequence the fragments in this pool.

The next thread would be responsible for receiving the fragments from the clients/end users, and adding them into a pool. Figure 7.2 gives a brief

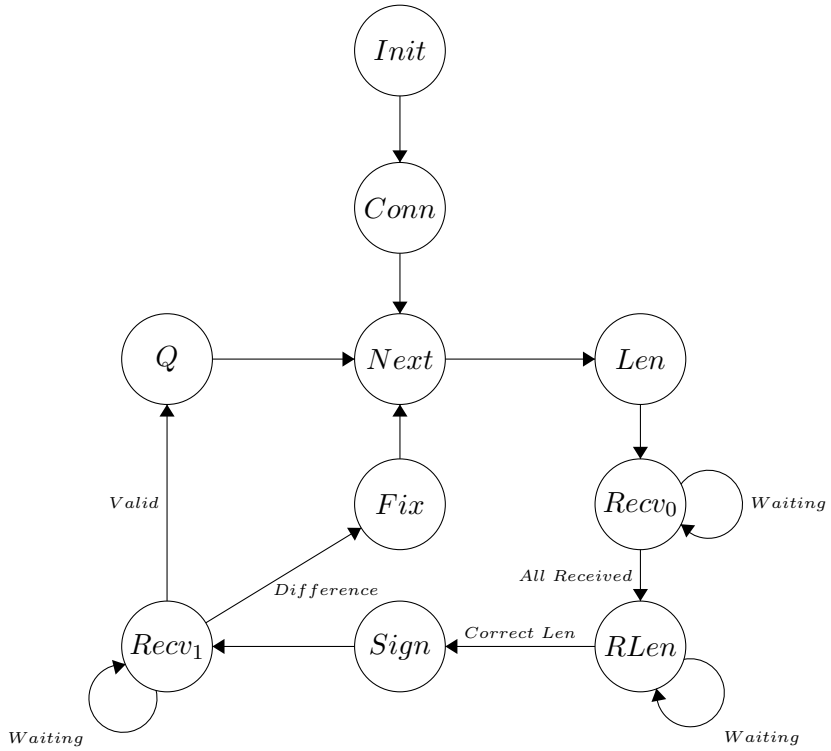

Figure 7.1: Flow for the scheduler to get the next fragment pool

overview, where it waits for a new user, receives the fragment value, and stores it as a key pair with some identification. The *Pool* state would check if the fragment server is managing the pool this fragment would go into. If it is the managing fragment server, it sends the index into the pool, the pool number, and fragment identification number to the other fragment servers. Otherwise, the server waits for a new user. Finally, Figure 7.3 will handle the receiving of indexes for the pools. Upon receiving an index and pool for a fragment identification number, the server checks to see if another fragment exists for that index in the pool. If that slot is empty, the fragment is added to the pool. Any errors here will be handled by the next pool thread.

With enough fragments being received and an even distribution between the pools, this simple scheduler will allow FRIBs to process fragments in order with no overhead of fragment identification, where after each addition the next fragment can be processed. This is because the cost of adding a new fragment to a pool is less than a reduction request. However, in order for the scheduler to not impact the performance of computation with FRIBs, it should be run

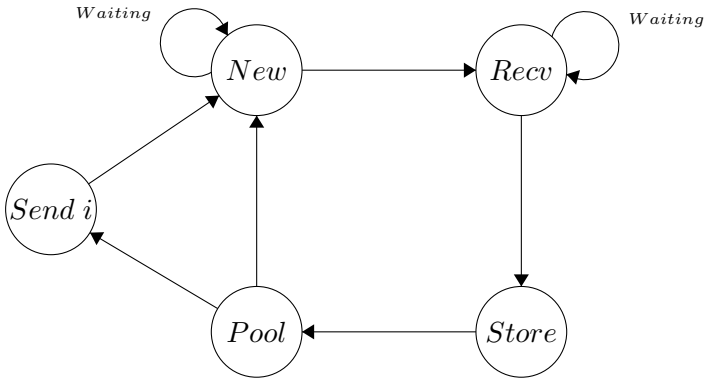

Figure 7.2: Scheduler subthread flow for receiving fragments from clients.

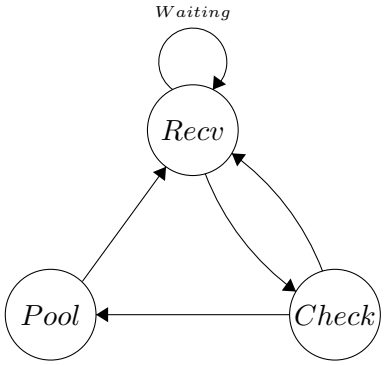

Figure 7.3: Scheduler subthread flow for receiving new indexes a pool

over a different ethernet stack/interface. Also, a consideration is available processing cores, where the scheduler threads should not impact or take away processing time from the FRIBs processing requirements.

7.1.2 Table Generation

To implement FRIBs, the LUTs need to be generated by a single entity. For an organisation storing and processing data in the cloud, they would generate their own tables. However, for processing across multiple entities, as it would be for government voting, another entity would be required to generate the tables. The performance of LUT generation is not critical to the performance of FRIBs, because they can be reused and only need to be generated once (they can be regenerated regularly to limit patterns, but each request can use the same LUTs).

Depending on the operation that is being implemented, only the function

that defines what result a combination of fragments represents needs changing. For example, Listing 7.1 defines the function to add a carry bit to x-bits using the XOR fragmentation algorithm. The argument *args* is an array of $N = len(args)$ fragment states, where the carry bit is at bit index 0 of each fragment. The resulting array contains the result fragments, where $N - 1$ can be random, and the last fragment is chosen so that the fragments give the correct result. Sample fragment input of $15, 10, 5$ ($01111 \oplus 01010 \oplus 00101 = 00000$) could give $30, 25, 7$ ($11110 \oplus 11001 \oplus 00111 = 00000$), or input $15, 10, 7$ ($01111 \oplus 01010 \oplus 00111 = 00010$) which is 0001 plus the carry bit 0, gives an output of $19, 4, 22$ ($10011 \oplus 00100 \oplus 10110 = 00001$), where bit index 4 is the resulting carry of the addition.

Listing 7.1: Function to get the result of fragments for addition

```
def getAddResult(args):
        r = []
        v = c = 0
        for arg in args:
                c = c ^ (arg & 1)
                v = v ^ (arg >> 1)
        v = v + c
        rv = 0
        for i in range(len(args)-1):
                r.append(random.randint(0,31))
                rv = rv ^ r[i]
        r.append(rv ^ v)
        return r
```

Adding redundancy makes the table generation more difficult, as different state combinations need to give the same result. Instead of randomly choosing the resulting fragments, the results need to be precomputed such that if corruption to one state occurs, the same result is returned if there was no corruption. The function for getting the fragments would be a lookup to get the precomputed result, as shown in Listing 7.2. The locations of sample code for generating the precomputed results are given in Appendix D.

Listing 7.2: Function to get the result of fragments for addition with redundancy

```
def getAddRedResult(args):
        if str(args) in validResults:
                return validResults[str(args)]
        return None
```

Another example of a resulting fragment function would be that of a NAND function with variable usage. This would be a result LUT that could be used to

compute nearly any arbitrary operation in a privacy-preserving manner. The LUTs can be computed before the algorithm or function has been defined, allowing for new functions or changes to existing functions without regenerating LUTs. If all the fragment servers are managed by different entities, as would be the case for voting, each entity can agree or disagree to running the new/-modified function. The process of getting the result fragments with a variable usage NAND function requires more steps than a static function. First, the state needs to be split such that the order of operations is observed, as shown in Listing 7.3. Recall that 0 is used to represent an operation and the number of 0s define order. For example, the state 1107 = 10001010011 would give $[1, [[1, 1], 11]]$, which in terms of NAND functions is $(1 \oplus ((1 \oplus 1) \oplus 0))$.

Listing 7.3: Function to split each fragment by NAND operation

```
def splitFragment(f, ops):
        if ops == 0:
                return f
        f1 = f.split('0' * ops)
        if len(f1) == 1:
                return splitFragment(f, ops - 1)
        output = []
        for sf in f1:
                output.append(splitFragment(sf, ops - 1))
        return output
```

With the order of operations preserved, the fragments need to be joined by the fragmentation algorithm (XOR). A sample is given in 7.4.

Listing 7.4: Function to join the fragments by the fragmentation algorithm

```
def fragmentEvalXor(f0, f1):
        if type(f0) is str:
                if f0 == "11" and f1 == "11":
                        return "11"
                if f0 == "1" and f1 == "1":
                        return "11"
                return "1"
        result = []
        for i in range(0, len(f0)):
                result.append(fragmentEvalXor(f0[i], f1[i]))
        return result
```

The fragment result can now be solved with Listing 7.5, giving a single bit. To implement a different operation, the NAND function can be replaced.

Listing 7.5: Function to split each fragment by NAND operation

```
def fragmentEval(f):
    if type(f) is str:
        if f == "11":
```

```
            return 0
        return 1
    e = fragmentEval(f[0])
    for i in range(1, len(f)):
        e = int(not(e and fragmentEval(f[i])))
    return e
```

Finally, the random result fragments, which are single bits, unlike the dedicated addition example, can be computed. Sample input $1107, 1577, 6355$ into the function for 7.6 would be a randomised fragment for $(1 \barwedge ((0 \barwedge 1) \barwedge 1)) = 1$, which could be $0, 1, 0$.

Listing 7.6: Function to get result fragments for variable NAND operations

```
def getNandResult(args):
        r = []
        rv = 0
        v = fragmentEval(reduce(lambda x,y: fragmentEvalXor(x,y
            ), map(lambda arg: splitFragment(bin(arg)[2:], 3),
            args)))
        for i in range(len(args)-1):
                r.append(random.randint(0,1))
                rv = rv ^ r[i]
        r.append(rv ^ v)
        return r
```

The remaining step for generating the LUTs is to generate a list of valid states, which in the case of the addition example in Listing 7.2 is $0 \rightarrow 2^{x-1}$, where x is the number of bits (including the carry bit). The generic NAND example requires validation because $29 = 11101$ would not be a reachable state. Once a list of states is generated, a random mapping for each fragment server to obfuscate its state for sending to the other fragment servers is generated (as described in Chapter 6). All possible state combinations can be tried and the resulting fragments can be generated using the functions defined earlier in this section. Finally, the obfuscation and result LUTs are outputted and distributed to their corresponding fragment servers.

7.2 Proof-of-Concept Voting Implementation

The comparison application and requirements for privacy-preserving computation used in Chapter 3 was electronic voting. This is due to the basic nature of voting being an addition, but it can also be made more complex by expanding to surveys and only allowing a certain number of votes for a set of options.

Two implementations are given in this section: the enhanced privacy model both with and without redundancy, requiring three and six fragment servers respectively.

7.2.1 Addition

Using the simple voting system introduced in Chapter 3, for the population of smaller countries such as New Zealand or Singapore, $2^{24} - 1$ would be large enough for every voter to vote *yes* for an option/candidate. Instead of adding two 24-bit numbers together, because a vote is a single bit, the system can be designed to add a single bit to a 24-bit number. This simplifies the adding process and allows for a customised addition operation to be defined. Smaller bit-sized values could be in tally pools, such that 4-bit values are used, but once 14 votes have been added to the tally pool value, it would need to be added to the final tally. This could help performance with parallelisation, but for comparisons with Chapter 3, a single 24-bit value will be used. The addition function for table generation in Listing 7.1 was used, such that the first x bits are added with the vote, before the next set of x bits are added with the carry bit from the previous operation. In this chapter, the enhanced privacy model without redundancy will have $x = 4$; therefore, six reduction requests are needed to add the vote to the 24-bit tally. The redundancy implementation sets $x = 3$ to reduce the size of the result LUTs and the LUTs sent over the network, meaning eight reduction requests are required.

With each set of x bits depending on the previous x bits, instead of adding the vote to the tally and then adding the next vote, they can be pipelined. Pipelining is common with hardware design, where stages of a function or operation are split over clock-cycles, such that the previous stage is processing the next segment of data. When the final stage is complete, after the next tick, the next segment has finished even though the actual function takes a number of clock-cycles. The reduction request can be treated as the clock-cycle, so each x bits are processing a different vote. Therefore, instead of a vote taking the time of six or eight reductions to be processed, the tally is actually updated every reduction, meaning if enough votes are queued, the input of a new vote and output of the final tally is achieved every reduction. This is shown in Figure 7.4, where after each reduction, a new vote has been

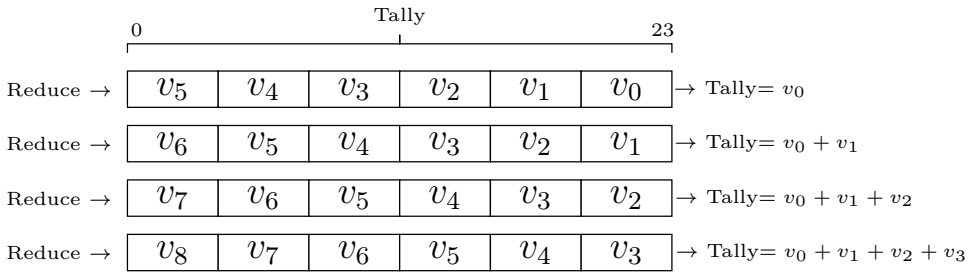

Figure 7.4: Reduction pipeline for adding votes to a 24-bit tally

added to the 24-bit tally; each block represents the x-bits, with the carry bit from the block on the left, but from the previous reduction. To get to the state shown in Figure 7.4, where v_0 is at the upper-most part of the tally, five reductions have already occurred. The first reduction shown in the figure is the sixth reduction overall and completes one addition.

7.2.2 Verification

Proving a vote is either *"yes"* or *"no"* comes free by only accepting a single bit from the voter (vote $\in \{0, 1\}$); however, there are other aspects of voting that may need verifying. With multiple candidates, there could be a need to guarantee only one vote has been cast, as discussed in Section 3.3.4 (this could also be seen as answers to a question in a survey). A sticky adder can be used to verify that at most one candidate receives a *"yes"* vote, by keeping the second bit high if it is ever set. Algorithm 6 is an example of a ballot consisting of x candidates, where each voter only gets a single vote. If $voter_a$ casts $[0, 1, 0]$, the function will return $[0, 1, 0]$, because the variable *sticky* does not get set. However, when $voter_b$ casts $[1, 0, 1]$ the *sticky* bit is set, meaning the resulting array is $[0, 0, 0]$. This nullifies the three votes from $voter_b$, because multiple candidates received a *"yes"*. The same nullification would happen with paper voting if multiple candidates are chosen. Another example for a ballot is where each voter gets three votes to use however they like. Now, each vote is multiple bits, instead of a single bit. In this case, the *sticky* bit would represent the third bit, so would only be set with four or more votes.

The concept behind Algorithm 6 can also be used for a ranking ballot:

for example, three available options, where the voter must pick an order.[1] In this example, there would be three separate ballots: first choice, second choice and third choice, giving nine tallies in total. For each choice, the votes are verified to only contain a single *"yes"* vote. Then, each option (option *A*s first, second and third choice) has the votes verified, meaning for the 2-dimensional array ($3x3$), a *high* bit can only have one bit set per column and one per row. The only change to the function is the *valid* bit gets applied to the whole 2-dimensional array. This does not stop a voter only choosing a first choice, but even with paper based voting this can still occur. If this was a requirement, then the logic for the *valid* bit on Line 7 of Algorithm 6 could be $valid \leftarrow \neg sticky \wedge add$, meaning that *add* must be set.

To record who has voted, the *valid* bit can also be stored. For each voter, their voter identification number would return 0, where the *valid* bit can be XORed with it to keep records of who has voted. This value can be combined with Line 7 of Algorithm 6, such that the vote is only valid if the voter has not submitted a valid vote previously. The voter identification number would be visible to each fragment server, but if a secure search could be implemented, the identification number could be fragmented and, therefore, kept private. The verification process was not tested in practice, because the performance of the addition is what is being compared to Chapter 3. However, all of the verification examples given can be easily implemented with FRIBs, where the XOR and *NOT* operations can be free (no reduction requests required). With the remaining operations being single bit functions, parallelisation for many voters could yield usable performance. A result LUT could even be generated to get the sticky bit of an array of votes (for each candidate), where the result is fragmented and therefore unknown.

Another requirement for a voting scheme would be that the voter can verify if their vote was cast correctly. This is out-of-scope of this use case, but worth discussing. A common technique is to use public bulletin boards [170][171], and that can be used here. When a voter sends their vote fragments to each fragment server, they can also send a random string (for each set of fragment). The voter also sends the random string to each public bulletin board and

[1] This type of approach would have allowed the New Zealand flag referendum to be implemented [169].

Algorithm 6 Verifying a single vote is cast

```
1: function verifyVote(votes)
2:     sticky ← 0
3:     add ← 0
4:     for v in votes do
5:         sticky ← sticky|(add ∧ v)
6:         add ← add ⊕ v
7:     valid ← ¬sticky
8:     for i in votes.length do
9:         votes[i] ← votes[i] ∧ valid
10:    return votes
```

receives back an index value. After the ballot (or periodically) the fragment servers can send their public bulletin board (one for each) the fragments with random strings. Note that the public bulletin boards should be hosted by different entities to the fragment servers as well. The public bulletin boards secretly decided together the index for each random string, which was sent back to the voter; therefore, after the ballot the voter can then verify their vote was cast correctly, by checking the index in the list of votes on the public bulletin boards. This also allows all the votes to be tallied to confirm the result. The votes and voter identification numbers are therefore kept separate. This is purely an idea and has not been explored for this thesis.

7.2.3 Enhanced Privacy Model

The implementation structures for the enhanced privacy models both with and without redundancy are nearly identical, with only a few differences for dealing with the different number of fragment servers. Both will be described in this section, but only the code for without redundancy will be given as examples, where both implementations can be found in Appendix D. Implementing FRIBs can be divided into three threads/processes: (1) performing the addition and reduction, (2) handling requests for another fragment server's result LUT, and (3) handling requests for correcting the obfuscation through indexes.

Performing the Addition and Reduction

The first stage of this thread is getting the next vote from the queue and setting it to be the first carry bit, where the remaining carry bits were set during the last reduction. In Listing 7.7, the *vote_window_len* will be six, as four bits plus the carry bit are reduced at once for the 24-bit tally. The carry bits are then appended to the list of states to be reduced.

Listing 7.7: Stage 1 for adding a new vote by getting it from the queue

```
carry_window[0] = self.voteaddQueue.get()
states = []
for i in range(vote_window_len):
        carry = carry_window[i]
        if carry == None:
                continue
        states.append(tally_window[i]  + (carry << 4))
```

The second stage involves obfuscating the fragment server's state for itself to use in the vector, and for handling requests from the other two fragment servers. Listing 7.8 gives an example of this process, where *qosb* and *qosc* are queues between the other two threads, as each thread needs to know the current state. Once the fragment server has obfuscated its own state (in *osa*) for use in the vector request, it generates three other random states to add to the vector, before shuffling it. Now that the vector is shuffled, the index of the correct obfuscated state is found and sent to fragment server C (via *sendc*). Note that each fragment server considers itself as fragment server A, so fragment server C is always the one to send the index too and to receive the new index and flip bit back from. The vector is sent to fragment server B (*sendb*) at the same time. This implementation is reducing the states in parallel and tries to fill the packets to improve performance. The only variable not covered so far is *vis*, which will be used later to recover the current index into the vector for that state.

Listing 7.8: Stage 2 for adding a new vote is to obfuscate the state

```
vis = []
sendb = sendc = ""
for state in states:
        osa = self.a.obfuscate(state)
        self.qosb.put(struct.unpack("<B", self.b.obfuscate(
            state))[0])
        self.qosc.put(struct.unpack("<B", self.c.obfuscate(
            state))[0])
```

```
        vec = [osa]
        i = 0
        while i < 3:
                r = struct.pack("<B", random.randint(0, 31))
                if r not in vec:
                        vec.append(r)
                        i += 1
        random.shuffle(vec)
        vi = vec.index(osa)
        vis.append(vi)
        sendb += vec[0]+vec[1]+vec[2]+vec[3]
        sendc += struct.pack("<B", vi)
self.b.ssl_sock.sendall(sendb)
self.c.ssl_sock.sendall(sendc)
```

The thread waits for the small obfuscated LUT from fragment server B, and the index and flip bit from fragment server C. Listing 7.9 shows the states being retrieved by selecting the correct row (using *vis* from earlier) and using the index and flip bit to get the result state.

Listing 7.9: Stage 3 for adding a new vote is to process the obfuscated LUT with the received index and flip bit

```
rstates = []
for si in range(len(states)):
        small_lut = []
        rows = self.b.ssl_sock.recv(128)
        row = rows[32 * (3 - vis[si]): 32 * (3 - vis[si]) + 32]
        for by in row:
                small_lut.append(by)
        index = struct.unpack("<B", self.c.ssl_sock.recv(1))[0]
        flip = struct.unpack("<B", self.c.ssl_sock.recv(1))[0]
        rstates.append(struct.unpack("<B", small_lut[index])[0]
            ^ flip)
```

The final step is to modify the *carry_window* for the next vote and update the tally (as shown in Listing 7.10). The thread then goes back to the first stage in order to add the next vote.

Listing 7.10: The final stage for adding a new vote is to set up the windows for the next vote

```
results = []
for i in range(vote_window_len):
        if carry_window[i] == None:
                continue
        carry_window[i] = (rstates[rsi] & 16) >> 4
        tally_window[i] = (rstates[rsi] & 15)
for i in range(vote_window_len-1, 0, -1):
        carry_window[i] = carry_window[i-1]
```

Handling Requests for Another Fragment Server's Result LUT

This thread is responsible for generating the small obfuscated result LUT for another fragment server. Note that for all the implementations and explanations in this thesis, the result LUT is for the lower fragment server; for example, fragment server B generates the LUT for fragment server A. With the three-server enhanced privacy model, this thread receives the randomised vector of states from the lower fragment server, and gets the possible results for each index in the vector, whereas the redundancy model will receive multiple vectors; in this implementation, it is five vectors (from all other fragment servers), in order to keep the data transferred to a minimum.

The simplicity of this thread can be seen in Listing 7.11. The thread receives four possible states from the lower fragment server. It then gets its own obfuscated state (*osc*) in order to get the result vector for each of the four states received. The result vector is shuffle, before a random value is XORed with it to hide the values. Once all four result vectors are obfuscated, they are sent back to the lower fragment server.

Listing 7.11: Loop which generates the obfuscated result LUT

```
while True:
        b4 = conn.recv(4)
        if len(b4) == 0:
                break
        osc = self.qosc.get() * 32
        rbs = ''
        for i in range(4):
                i = (struct.unpack("<B",b4[i])[0] * 32 * 32) +
                    osc
                row_bytes = self.result_lut[i:i+32]
                self.c.random.shuffle(row_bytes)
                for rb in row_bytes:
                        tmp_r = self.c.random.randint(0,31)
                        rbs += struct.pack("<B", struct.unpack(
                            "<B", rb)[0] ^ tmp_r)
        conn.sendall(rbs)
```

Handling Requests for Correcting the Obfuscation through Indexes

The final thread provides the upper fragment server (for example, at fragment server B, the upper fragment server is C) with the index for the small obfuscated result LUT that is generated by the previous thread. The thread receives the index for the vector sent by the upper server, where the index is

for the position of the correct state (as only one state in the vector is correct). Using this index and its own obfuscated state, it can generate the index for the new state of the upper fragment server. Again, the redundancy model would receive more indexes, as five vectors are sent to generate the obfuscated LUT.

To further explain, Listing 7.12 contains the loop which is responsible for generating the index value. It waits to receive the index value, then gets its own obfuscated state value. The thread must handle the random shuffle and numbers generated by the previous thread to keep the pseudo-random number generators aligned. In this implementation, the approach to solve this is very basic; it first shuffles an empty array of the same size, then for each state in the result vector it generates a random number. Recall that this thread shares the same seed for the pseudo-random number generator as the thread generating the small obfuscated result LUT. Once the correct index is reached, the index and the random value is saved to send back to the upper fragment server. Note that it cannot be sent once it is found to prevent timing attacks [172].[2] The random value is the flip bit mentioned previously, to reveal the new state.

Listing 7.12: Loop which generates the obfuscated index

```
while True:
        b1 = conn.recv(1)
        if len(b1) == 0:
                break
        ri = struct.unpack("<B", b1)[0]
        osb = self.qosb.get()
        osbf = 0
        for i in range(4):
                row_bytes = []
                for j in range(32):
                        row_bytes.append(j)
                self.b.random.shuffle(row_bytes)
                osb_new = row_bytes.index(osb)
                for j in range(32):
                        tmp_r = self.b.random.randint(0,31)
                        if i == ri and j == osb_new:
                                osbf = tmp_r
                                osbi = osb_new
        conn.sendall(struct.pack("<B", osbi) + struct.pack("<B"
            , osbf))
```

[2]Timing attacks are where a function takes a varying amount of time for different inputs [172], thus potentially revealing the index.

7.2.4 Performance

To be able to claim practical performance for the addition aspect of voting, the following question must be true.

Question 3 *Can all eligible voters cast their vote during the voting period, and the result be available shortly after the ballot is finished?*

Definitions of some of the terms in the question above can vary, where scope is needed. For this analysis, the voting period is 24 hours, but the majority of the votes will be cast during a 12-hour period, for a voting population of around five million. The distribution of votes cast will greatly affect when the results are available, because if all votes are cast in the last hour, the results cannot be expected to be finalised when the voting period ends. Therefore, if all votes are cast at 5am, the tally should be complete by 5pm, reducing analysis to throughput (votes per second). All results in this section were from a cloud implementation in AWS using *C5.large* instances, consisting of 2 virtual central processing units, 4GB memory, and up to 10Gbps network performance.

For the enhanced privacy model consisting of three fragment servers, the LUT sizes for both results and a single obfuscation LUT were 32KB and 32B respectively. In terms of memory requirements, these tables can fit easily into main memory and higher levels of cache. By doubling the number of fragment servers, the redundancy model result LUT is 16MB, but the obfuscation LUTs drop to 16Bs, as the number of states has halved (only reduces three bits plus a carry, whereas without redundancy, the reduction occurs on four bits plus the carry); however, these tables will still be able to reside in memory, meaning the memory requirements of the two implementations are very small.

The biggest overhead for any FRIBs implementation will be the network, including latency, bandwidth, packet loss and the cost of moving data to/from the networking stack. This is shown in Figure 7.5 where different RTT times are compared against multiple voting tallies (parallel tallies) for the time taken to process a single vote. Focusing on the single 24-bit tally, the difference between 0ms RTT (actually 0.05ms) and 1ms (1.05ms) is practically nothing; therefore, the time to process a single vote is primary processing time on the central processing unit and transferring data to/from the networking stack.

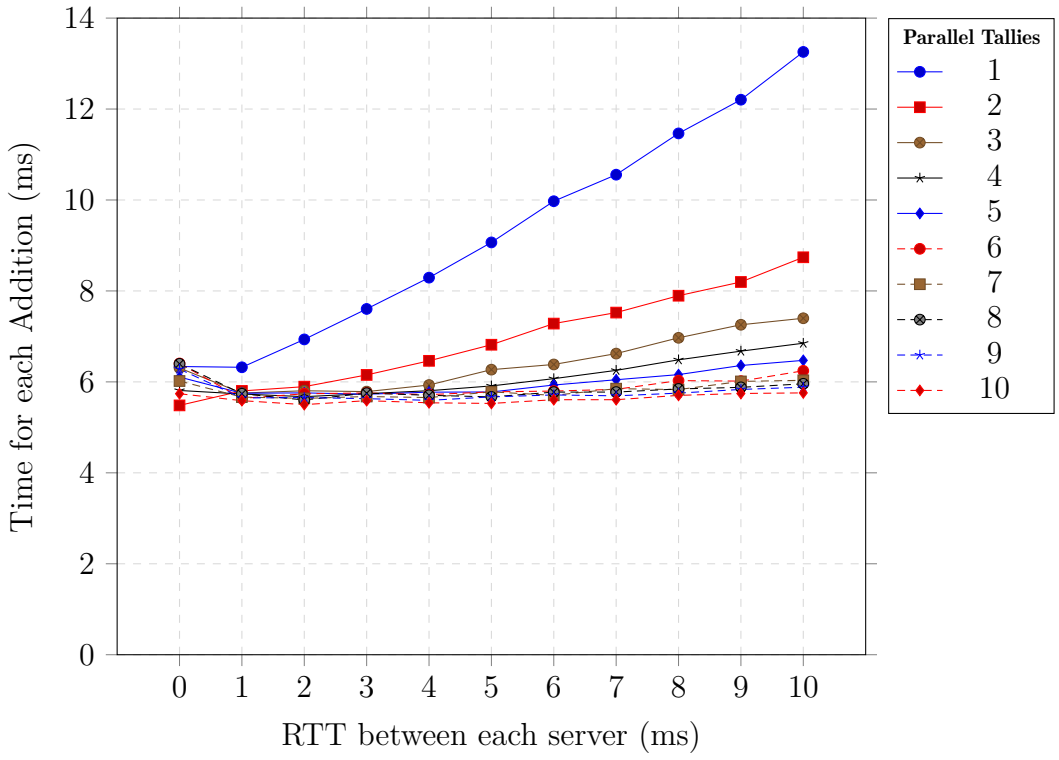

Figure 7.5: Votes added per second, for a three-server model

However, once more latency is added, the time to process a single vote increases at a near linear rate. Computing multiple 24-bit tallies in parallel hides a lot of the networking overhead, as the other data lines in Figure 7.5 show. An example is ten parallel tallies, where the time to add a single vote stays consistent even with varying latencies. The overhead for the network would therefore just be bandwidth availability and packet loss. Note that the difference between sending the data over a secure channel or unsecured channel did not affect throughput in a noticeable manner.

Adding a vote to the single 24-bit tally takes 0.013 seconds for a 10ms RTT between each fragment server, giving a throughput of 75.4 votes per second. Within a 12-hour period, over 3 million votes could be tallied–a nearly acceptable rate. However, with ten tallies in parallel, the throughput increases to 173.7 votes per second, allowing for over 7.5 million votes to be tallied in 12 hours. This is an acceptable throughput for the use case of New Zealand or Singapore. Once the tallies are complete, the ten tallies can be summed together within a FRIBs environment or in plain text. Note that by parallelising

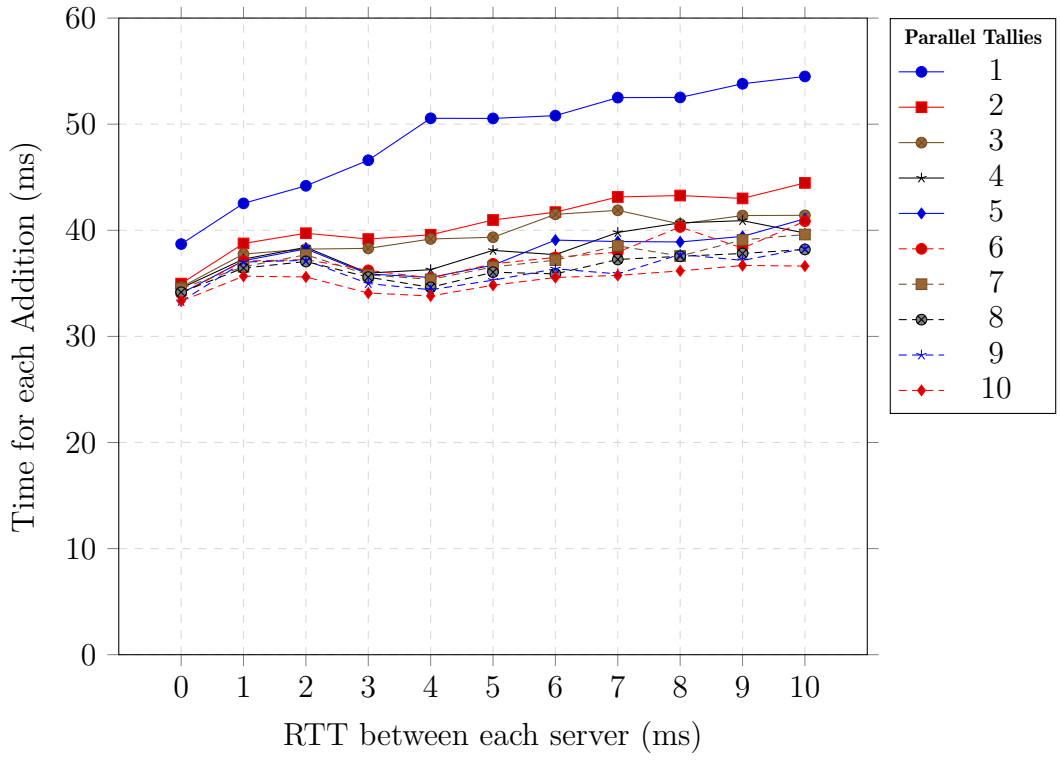

Figure 7.6: Votes added per second, for a six-server model with built-in redundancy

multiple tallies, the fragment servers wait for ten vote fragments in the queue, then reduce the ten tallies in parallel to saturate the network. The aim is to prevent the threads from waiting (going to sleep) while new data arrives from the network. The number of parallel tallies will vary on implementation and network quality between the fragment servers.

The redundancy model results for the time to process a single vote are given in Figure 7.6. Similar results to Figure 7.5 are observed; however, with more servers and more data transferred, the throughput is less.[3] The maximum throughput achieved was 30 votes per second, meaning only 1.3 million votes could be summed in twelve hours. For other application or scenarios, this performance would be acceptable, especially because it has built-in redundancy,

[3]Figure 7.6 has a small anomaly for RTT values between 1 and 4. The results were obtained by averaging additions for all x parallel tallies for each RTT, meaning each tally is running at the same time per RTT; therefore, any variation in the network at that time could cause the hump seen for RTT values 1 and 2. Instead of averaging the results over multiple days to remove this hump, the results were left as a single run to demonstrate how the network can affect performance.

but with the voting use case, the enhanced privacy model with redundancy is too slow for practical performance. Instead, it would be better to run three enhanced privacy model instances, giving nine fragment servers in total. The performance would be superior and also allow for one instance to be compromised, as the other two instances would give the same result.

In Chapter 3, partially homomorphic encryption was shown to give the best performance for secure voting in the cloud. Performance for the homomorphic addition taking under a millisecond gives unmatchable throughput value; however, because the zero-knowledge proof is required to guarantee votes are a single bit, it also must be included in the throughput calculation. Re-evaluating the performance of the partially homomorphic encryption scheme on the same AWS instance (one of the fragment servers used for testing in this chapter) gives a result of 22ms for a 2048-bit key, and 117ms for a 4096-bit key. Given the instances are dual-core, throughput achieved is 91 or 17 votes per second, depending on key size. Using the same number of servers as the enhanced privacy model results in six tallies, two per server with a throughput of 273 or 51 votes per second. Therefore, using the same compute power as the FRIBs implementation, partially homomorphic encryption can still produce a greater throughput; however, the client-side performance for generating the zero-knowledge proof is still an issue, especially within web browsers. However, FRIBs has client-side performance similar to that of plain text, and when combined with the flexibility offered by FRIBs (allowing for different operations to be computed, including advanced voting specifications), it compensates for the slightly slower throughput.

7.3 Proof-of-Concept Search Implementation

In Chapter 4, an approximate string searching scheme was presented and will be extended in this section with FRIBs. The previous section detailed a custom function for the result LUTs, and can only be used for adding a single bit to an integer. Therefore, in this section, a generic NAND result LUT will be used, allowing for arbitrary functions to be defined. This section will also use the performance model, as search operations will not occur as regularly as adding votes for millions of people. Up until this section, practicality has been defined

as performance (both for client and in the cloud), but this section will look at another aspect of practicality: flexibility and utility. Providing a service where the LUTs need to be generated for every execution or different function can limit the usage of the data. For example, data is hosted in the cloud but other organisations would like to run a custom function over the data. The organisations managing the fragment servers can verify and accept to run the function, without the need to verify the LUTs.

7.3.1 Secure Searching Use Case

A challenge for law enforcement agencies is keeping their investigations private, but also having the ability to share information. INTERPOL helps to facilitate this collaboration between its member countries [173]; however, there is still the issue of trust and privacy around information sharing. For example, if an entity hosts information regarding cryptocurrency, then ideally the other entities can search, view and download this information, without revealing what information they were accessing.

The design of INTERPOL's in-house developed tool "Bitcoin Explorer" is not relevant; however, the secure searching aspect–a proof-of-concept design–will be described. In this use case, there are some suspicious Bitcoin wallet IDs–a unique string–which a country would like to flag without revealing the ID to anyone else. The country can provide some details on the ID, which is only revealed when the ID is matched. When combined with anonymous routing [141], the country and wallet ID remains hidden. With FRIBs, a few member countries and INTERPOL can each host a fragment server, thus ensuring no single entity has access to the data.

7.3.2 Building FRIBs Functions in Lisp

Common Lisp was chosen as the implementation language of FRIBs for NAND functions in this thesis, because functions are first class objects, and it is designed for lists (values are a list of fragments). This makes defining programs for FRIBs seem natural, without knowing the underlying FRIBs scheme. For example, the function + can be redefined to execute an addition in FRIBs. Local testing can then be run on plain-text values, before being submitted to

the fragment servers where + is redefined to automatically translate the program to run in the FRIBs environment, as shown in Listing 7.13. This makes development of algorithms/programs simpler and means no custom compiler is required. Note that for a performance optimised implementation, the approach for voting in Section 7.2 is recommended where custom LUTs and threads are defined.

Listing 7.13: Redefining the addition function in Lisp

```
(unlock-package 'common-Lisp)
(unlock-package 'common-Lisp-user)
(defconstant +old-plus+ (fdefinition '+))
(defun local+ (&rest args) (apply +old-plus+ args))
(defun + (a b) (fribs-add a b))
```

The basic structure for a fragment is given in Listing 7.14, where *fragment* is the fragment value (1101), *len* is the bit length, and *num* is the number of operations applied to this fragment value. The last two variables are used to improve performance, but are not necessary.

Listing 7.14: Fragment structure

```
(defstruct frib
  fragment
  len
  num
)
```

Two NAND functions are defined in Listings 7.15 and 7.16. The first is a generic function with no optimisations, where the second function also gets passed (as an argument) the number of operations to set the operation value faster (for example, 000 is the third layer of operations).

Listing 7.15: Lisp NAND function definition

```
(defun nand(a b)
  (prog (o)
    (setf o (make-frib :fragment 0 :len 0 :num 0))
    (setf (frib-num o) (+ 1 (max (frib-num a) (frib-num b))))
    (setf (frib-fragment o) (logior (frib-fragment a) (ash (
       frib-fragment b) (+ (frib-len a) (frib-num o)))))
    (setf (frib-len o) (+ (frib-len a) (frib-num o) (frib-len b
       )))
    (return o)
  )
)
```

Listing 7.16: Lisp optimised NAND function definition

```
(defun nandc(a b c)
  (prog (o)
    (setf o (make-frib :fragment 0 :len 0 :num c))
    (setf (frib-fragment o) (logior (frib-fragment a) (ash (
       frib-fragment b) (+ (frib-len a) (frib-num o)))))
    (setf (frib-len o) (+ (frib-len a) (frib-num o) (frib-len b
       )))
    (return o)
  )
)
```

The base functions have now been defined, allowing other basic functions to be built upon them, with some given in Listing 7.17.

Listing 7.17: Lisp functions for NOT and AND gates

```
(defun fribs-not(a)
  (nand a a)
)
(defun fribs-notc(a c)
  (nandc a a c)
)
(defun fribs-and(a b)
  (fribs-not (nand a b))
)
(defun fribs-andc(a b c)
  (fribs-notc (nandc a b (- c 1)) c)
)
```

The basic operations grow the fragment value; therefore, a reduction function is required. The reduction function in Listing 7.18 is for the performance model but can be redefined to use the enhanced privacy model. The function in Listing 7.18 first obfuscates its states for the other two fragment servers and sends the value to the network sockets (*st_ob* and *st_oc*). It then waits to receive the obfuscated states of the other fragment servers before getting its new state.

Listing 7.18: A reduction function for the performance model

```
(defun reduction(fs)
  (prog (frag data bytedata a b c)
    ; Send fragments to the other servers
    (loop for f in fs do
      (setf frag (frib-fragment f))
      (setf b (mod (position frag *table_b*) 64))
      (setf c (mod (position frag *table_c*) 64))
      (write-byte (logand 255 b) *st_ob*)
      (write-byte (logand 255 c) *st_oc*)
    )
    (force-output *st_ob*)
    (force-output *st_oc*)
```

```
    ; Get reduced fragments
    (loop for f in fs do
      (setf frag (frib-fragment f))
      (setf a (position frag *table_a*))
      (setf b (read-byte *st_ib* nil))
      (setf c (read-byte *st_ic* nil))
      (setf data (aref *table_abc* (+ (* a 4096) (* b 64) c)))
      (setf (frib-fragment f) data)
      (setf (frib-num f) 0)
      (setf (frib-len f) (integer-length data))
    )
  )
)
```

7.3.3 Matching Operation

When applying an XOR operation over two binary values, if resulting bits are all *low*, the values must be identical. A single matching bit can be obtained by ORing each bit together and flipping the result. An example is shown below, where a and b are compared, with the result stored in m. For multiple values, combining the match result with $m = m \vee m_i$ will keep track of whether a match was found. Therefore, the fragment servers have no knowledge if a result was found, or the data that is being compared. The XOR operation is provided for free by FRIBs when using XOR as the fragmentation algorithm.

$$c_{0-31} = a_{0-31} \oplus b_{0-31}$$
$$m = \neg(c_0 \vee c_1 \vee ... \vee c_{30} \vee c_{31})$$

The OR function can be defined similar to Listing 7.19, where an array of bits are ORed together to produce a single bit. Essentially, the function checks if any bit is *high* for a. This function highlights a limitation for the LUTs that were generated, where they need the fragment to have six operations applied. The variable **frib-one** represents a *high* bit value, and is set during configuration. For three fragment servers and the XOR fragmentation algorithm, each **frib-one** can be set to 1.

Listing 7.19: An OR function in Lisp for FRIBs that process an array of bits, returning a single bit

```
(defun fribs-1-or(a)
  (prog (i j tmp tmp2)
    (setf i (list-length a))
    (setf tmp a)
```

```
    (loop while (> i 1) do
      (setf tmp2 (list))
      (loop for j from (- i 1) downto 0 by 2 do
      (setf tmp2 (append tmp2 (list (nandc *frib-one* (nandc *
         frib-one* (nandc (nandc (nth j tmp) (nth j tmp) 1) (
         nandc (nth (- j 1) tmp) (nth (- j 1) tmp) 1) 2) 2) 2))
         ))

      )
      (reduction tmp2)
      (setf tmp tmp2)
      (setf i (/ i 2))
    )
    (return (nth 0 tmp))
  )
)
```

The overall result bit can then be set by defining an AND function that includes the reduction, as shown in Listing 7.20. Again, this function needs to make the fragment value have the required number of operations for the result LUTs. The matching bit for this entry is then ANDed against the information stored for that entry, and another AND function is defined to handle this, given in Listing 7.21.

Listing 7.20: An AND function in Lisp for FRIBs that includes the reduction step

```
(defun fribs-and2(a b)
  (prog (o)
    (setf o (list (nandc *frib-one* (nandc *frib-one* (
       fribs-and a b) 2) 2)))
    (reduction o)
    (return (nth 0 o))
  )
)
```

Listing 7.21: An AND function in Lisp for FRIBs where a single bit is ANDed against an array

```
(defun fribs-and3(a b)
  (prog (o)
    (setf o (list))
    (loop for i from 0 to (- (list-length b) 1) do
      (setf o (append o (list (fribs-andc (nth i b) (fribs-notc
          a 1) 2))))
    )
    (reduction o)
    (return o)
  )
)
```

The functions for addition and multiplication can be found in Appendix D, but are the same as described in Section 6.3.

7.3.4 Search Implementation Example

The example implementation for string searching in FRIBs built from NAND functions included a matching value and a detailed information field. The matching value was a 32-bit integer, but for the use case in Section 7.3.1, it would be a bitcoin address. The details field had a character limit of 100, with padding to hide length. Each fragment for the detailed information can be ANDed with the matching value, before being ORed with the resulting detailed information. Therefore, only the matching fields' detailed information is set. This also means that the same address cannot be tagged twice, but information would need to be appended to the existing entry. Listing 7.22 gives the searching algorithm, using the functions previously given, where the value a is compared against a list of stored values in *frags*.

Listing 7.22: Searching for the 32-bit integer a in a list of stored values

```
(setf r 0)
(setf ro 0)
(setf rt (list))
(loop for i from 0 to (- (list-length frags) 1) do
  (setf ro (fribs-1-or (fribs-xor (nth i frags) a)))
  (if (> i 0)
    (setf r (fribs-and2 r ro))
    (setf r ro)
  )
  (loop for j from 0 to 99 do
    (if (= i 0)
      (setf rt (append rt (list (fribs-and3 ro (nth j (nth i
         text))))))
      (progn
        (setf fa3 (fribs-and3 ro (nth j (nth i text))))
        (loop for k from 0 to 31 do
          (if (= (frib-fragment (nth k (nth j rt))) (
             frib-fragment (nth k fa3)))
            (progn (setf (frib-fragment (nth k (nth j rt))) 3)
               (setf (frib-len (nth k (nth j rt))) 2))
            (progn (setf (frib-fragment (nth k (nth j rt))) 1)
               (setf (frib-len (nth k (nth j rt))) 1))
          )
        )
      )
    )
  )
)
```

The implementation example is given in Appendix D; with an example of the implementation running given in Figure 7.7 (a larger version of this screenshot is given in Appendix C). The client is shown in the upper-left terminal, and the

fragment servers are given in the other three terminals. The time for the client to send and retrieve the result for one stored value with this implementation was 0.540 seconds, with a RTT of 5ms between each of the three fragment servers and the client. This is not ideal, because as the number of stored values begins to grow, the performance of this design will decay, as each entry needs to be compared, and even if it is not a match, the resulting details also need to be computed. Parallelisation is a possibility using a divide-and-conquer approach, or the results from different threads can be combined. However, Bin Encoding presented in Chapter 4 could be used to provide a subset of entries to search over. Bin Encoding itself is a best effort searching scheme, but it can be combined with FRIBs to provide accurate results. There are issues raised

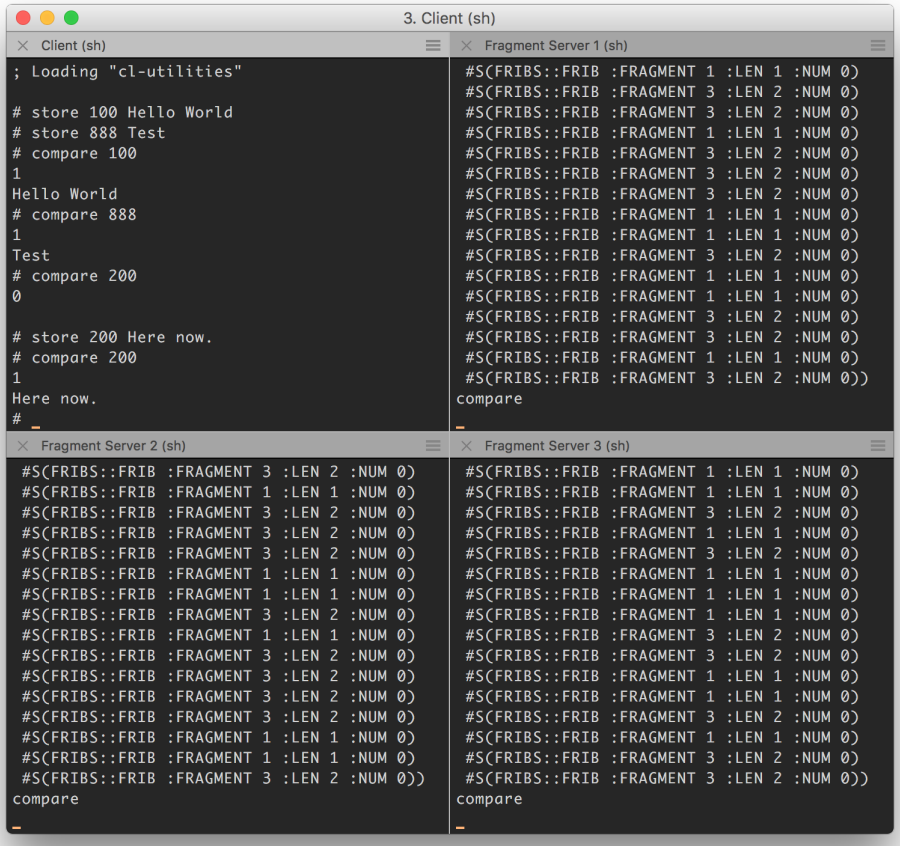

Figure 7.7: A proof-of-concept screenshot of searching for tagged wallet ID

with this proposal; for example, if a Bin Encoded query has no matches to begin with, then the servers know that no match was found. However, if Bin Encoding can give a subset of ten entries from thousands of stored values, then 5.4 seconds to securely search thousands of records is acceptable. A dedicated design like voting in Section 7.2 would yield faster results, but the purpose of this section was to show how a universal function such as NAND could build more complex functions in FRIBs.

7.4 Advanced Security Analysis

For this section, the enhanced privacy model will be used unless stated otherwise. It offers the best privacy out of the models presented in this thesis, while providing practical performance for applications such as voting.

7.4.1 Frequency Analysis Attacks on Repeating States

For the performance model, frequency attacks are viable, because each server can see the obfuscated states of the other server; however, with the enhanced privacy model, a configuration of three fragment servers and using a fragmentation algorithm with all XORs, a frequency analysis attack becomes very difficult. For example, with Table 7.1, there is an equal probability for any fragment value; even for a *high* or *low* result, there is an equal probability of a fragment server receiving a 0 or 1. A frequency analysis attack becomes more viable when an entity has multiple fragment values. However, decoding the real value depends heavily on having all the fragments. For example, with knowledge of F_B and F_C in Table 7.1, there is still an equal probability of the result being *high* or *low*.

During processing, each server has no knowledge of the others state, apart from a vector list from one other server. The reduction flow example given in Figure 5.14 would not be vulnerable, because all states are requested. Therefore, there is a 100% chance each state appears in the vector; however, if only a subset is requested, then a frequency attack becomes viable. This attack would look for a server repeating states continuously, or at a certain point in the program. Figure 7.8 shows 100 reduction steps, where the server was always in the obfuscated state 0. The vector contains any of the 32 states in

Table 7.1: Fragment options for three servers using XOR

F_A	F_B	F_C	Result
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

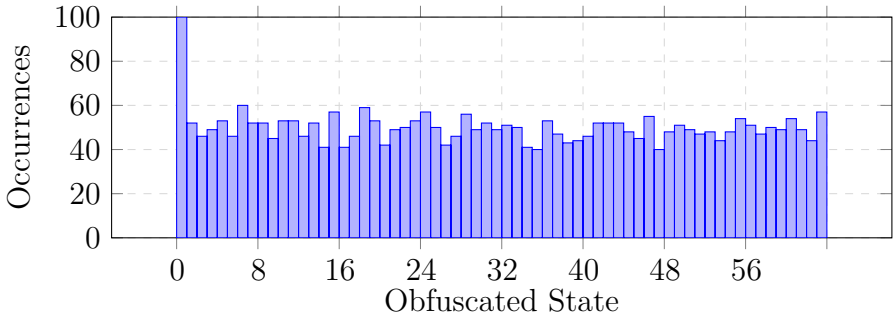

Figure 7.8: Occurrences of 32 states in a vector where the server is in obfuscated state 0 for 100 times in a row

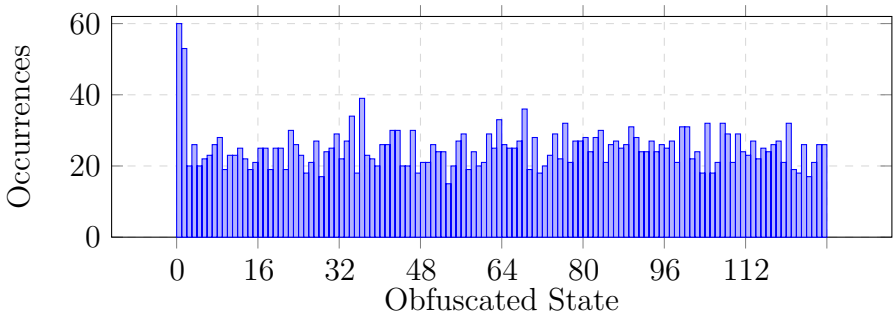

Figure 7.9: Occurrences of states in a vector when the server state has two obfuscated states, in this case 0 and 1 are the same state and repeated 100 times in a row

random order; therefore, state 0 is in the vector 100 times–standing out from the others. If two obfuscated states per single concatenated fragment are used, they still occur more often than the others, as shown in Figure 7.9.

A solution to stop a single obfuscated state from occurring more often than any other is to group the states together, meaning if state 0 occurs, so does

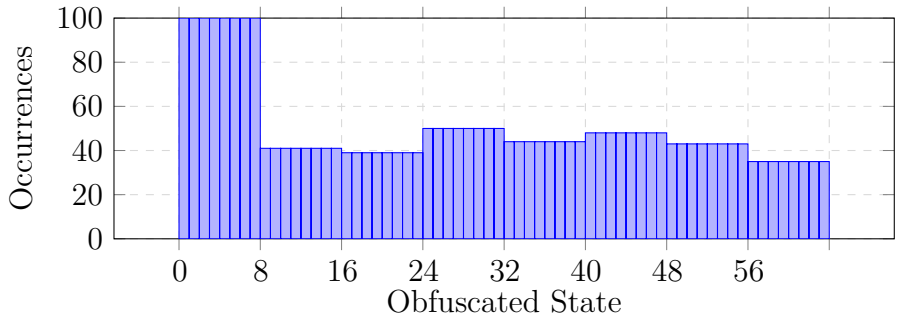

Figure 7.10: Occurrences of states in vector when the server in obfuscated state 0 for 100 times in a row, where obfuscated states are in groups

another set of states. In Figure 7.10, states are grouped together in groups of eight. Again, state 0 is reoccurring and the vector sends 32 states. Here, eight states occur more often, making it more difficult to learn what state is reoccurring, or if it is switching between states in that group.

If the states are changing, it becomes less clear which states are being reached. Figures 7.11, 7.12 and 7.13 randomly switch between obfuscated states $0-3$, $0-7$ and $0-15$ respectively. Even with only four states being reached, the states are starting to become hidden within the states that are not occurring. Once 25% of the states are being reached randomly, they are completely hidden. Therefore, if fragments' values are changing, as long as there are a few different states being reached, they are hidden in the vector. With the randomisation that can occur by a fragment server flipping two of the three resulting bits, as discussed in Section 6.5.1, the resulting states could vary even if all fragment servers had the same state. To summarise, the probability of a fragment being 0 or 1 is theoretically equal, and states will be hidden in the requested vector if states change.

7.4.2 Analysis for Voting Implementation

In this section, the security of the three-server enhanced privacy model is analysed with randomisation enabled (described in Section 6.5), to prove that FRIBs does not leak any information about votes and the final tally. This will be done by recording the tallies' state and transitions to see if there are any differences based on the vote. If all the votes for a tally are *no*, then there should not be any bias in which states the fragment server reaches while

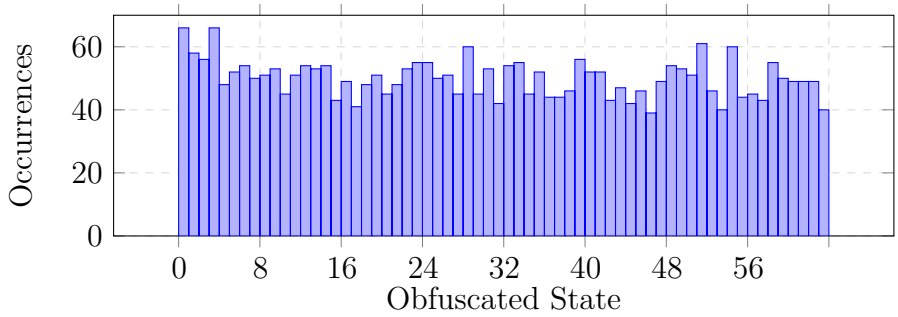

Figure 7.11: Occurrences of states in 1/2 vector when the server is in the obfuscated state 0, 1, 2, or 3 for 100 times in a row

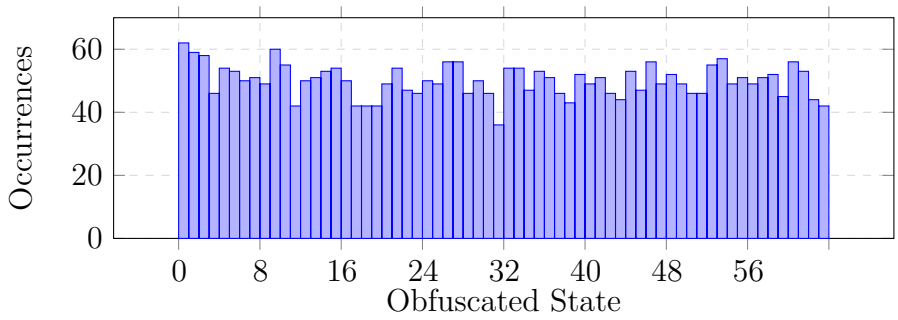

Figure 7.12: Occurrences of states in 1/2 vector when the server in obfuscated state $0-7$ for 100 times in a row

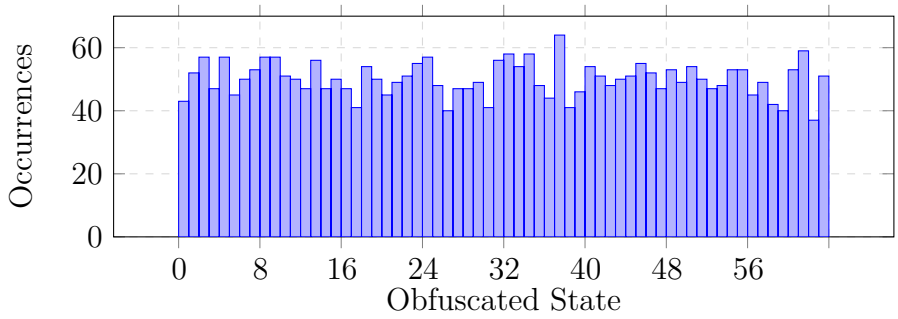

Figure 7.13: Occurrences of states in 1/2 vector when the server in obfuscated state $0-15$ for 100 times in a row

processing. Figure 7.14 gives the states reached when adding thousands of *no* votes with random fragments. There is no obvious bias to any one state, and furthermore, there was no bias between states with an even or odd number of bits. Figure 7.15 gives the state transitions from state 1 (00001_2) and again there is no bias between states (including even or odd number of bits). The same results are seen in Figure 7.16 and Figure 7.17 for random votes, where

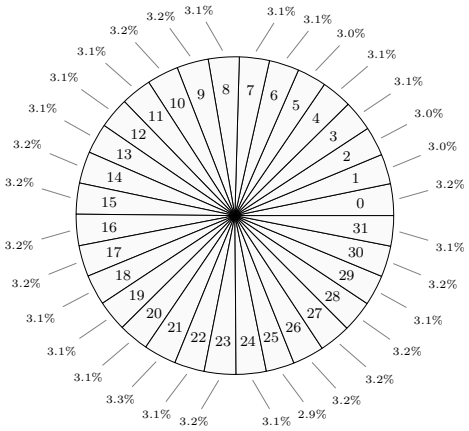

Figure 7.14: Percentage for each state reached with all votes cast as no (zero)

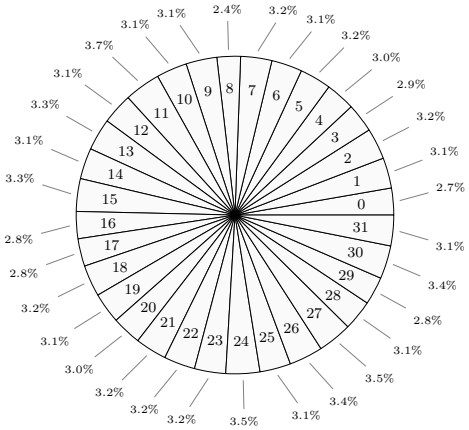

Figure 7.15: Percentage of states reached after state 1 for all no votes

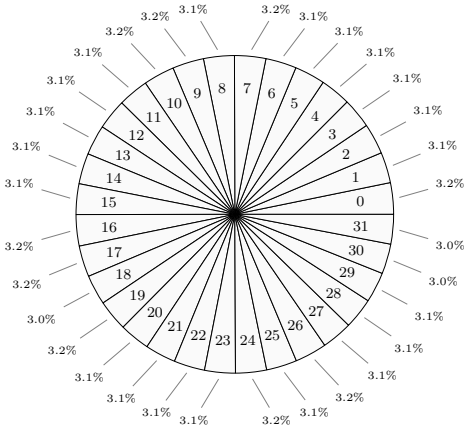

Figure 7.16: Percentage for each state reached where all votes cast are random

there is no bias for any state (Figure 7.16), or transition (Figure 7.17).

The final analysis for the voting implementation is recording the states received in the vector (from the lower fragment server) and which state the

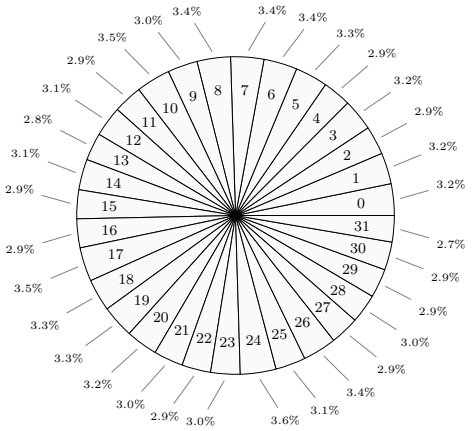

Figure 7.17: Percentage of states reached after state 1 with random votes

fragment server is in. Figures 7.18 and 7.19 show the distribution of states in the vector of size four received from fragment server C when fragment server A was in state 1, for all *no* votes and random votes respectively. These results show there is no favouritism towards any state, where with enough runs the results would be theoretically equal. The results in this section show that even a simple XOR fragmentation function with fragment servers randomly flipping bits, can provide enough randomness to keep the tally protected.

The other attack vector is brute-force; continuously guessing which state the other fragment servers are in. A fragment server knows its own state, and four possible states of another, but has no information about the remaining fragment server. This gives 4×32 combinations for a 4-bit value; however, knowing which is correct is improbable. Note that the four states received in the vector are obfuscated (a random mapping), so they do not directly reveal the real state. There are $32! \equiv 2.63 \times 10^{35}$ possible mapping combinations, so unless a large amount of data is being attempted to be broken, it is more efficient to guess the state, giving 32×32. Therefore, the 24-bit tally value would have $(32 \times 32)^6$ combinations, where again it is improbable to know when the correct combination is found. With 10 tallies in parallel, there would be $(32 \times 32)^{60}$ overall combinations for the system (4.15×10^{180}). In this case, using the mapping combinations produces fewer overall combinations at $32! \times (4 \times 32)^{60}$ (7.12×10^{161}). The enhanced privacy model with redundancy does not provide the same number of combinations; even with ten parallel tallies, it gives $(32!)^2 \times (4 \times 4)^{60}$ (1.22×10^{143}), because each fragment server receives at least two vectors, and only three values are needed for a decoding (including its own

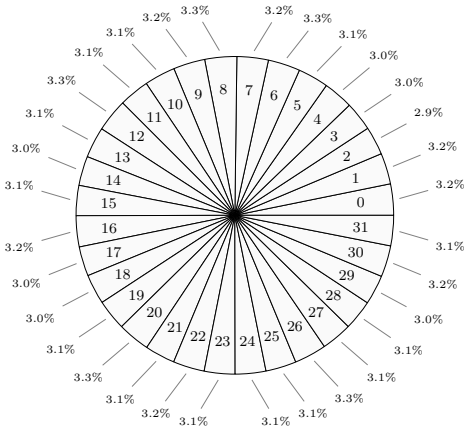

Figure 7.18: Percentage of states received in the vector when the fragment server is in state 1 for all no votes

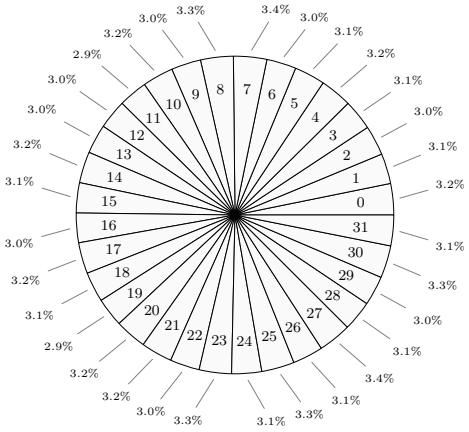

Figure 7.19: Percentage of states received in the vector when the fragment server is in state 1 for all random votes

state). It is possible to reduce the number of combinations by the number of voters, but the challenge is knowing when the correct combination is found, even with just 4-bits and only fifteen voters. Unlike encryption, solving one of the $4-bit$ values does not reveal the other values, meaning with megabytes of data to try and solve, the number of overall combinations is similar or greater than brute forcing encryption schemes. In summary, the enhanced privacy model gives the best security properties for FRIBs, whereas the redundancy model provides an additional feature, sacrificing some performance and security.

7.4.3 Two Fragment States

To gain an understanding of the security for the state obfuscation and random mappings, a three-fragment server model for the reduction server and performance model will be analysed to see what can be learnt from the states received. The fragmentation operation is again known to be XOR. This section will be analysing how many possible mapping options there are for a fragment or concatenated fragment to their corresponding obfuscated state.

7.4.3.1 Two States

We will start with two states (meaning no operation can actually occur), where $fragments = \{1, 11\}$ and $states = \{0, 1\}$. The example of reducing fragments of size 1, allows only two possible options for the data, if the order (the fragments are reduced in order, meaning f_0, then f_1, through to f_31 for a 32-bit integer) and operation is known. Comparing two sets of fragments i and j during a reduction stage gives the following set of rules. In this example A, B and C represent different fragment servers, and R is the resulting value.

Rules for two obfuscated states:

1. **if** $A_i = A_j$ & $B_i = B_j$ & $C_i = C_j$ **then** $R_i = R_j$
2. **if** $A_i \neq A_j$ & $B_i \neq B_j$ & $C_i = C_j$ **then** $R_i = R_j$
3. **if** $A_i \neq A_j$ & $B_i = B_j$ & $C_i \neq C_j$ **then** $R_i = R_j$
4. **if** $A_i = A_j$ & $B_i \neq B_j$ & $C_i \neq C_j$ **then** $R_i = R_j$
5. **if** $A_i \neq A_j$ & $B_i \neq B_j$ & $C_i \neq C_j$ **then** $R_i \neq R_j$
6. **if** $A_i \neq A_j$ & $B_i = B_j$ & $C_i = C_j$ **then** $R_i \neq R_j$
7. **if** $A_i = A_j$ & $B_i \neq B_j$ & $C_i = C_j$ **then** $R_i \neq R_j$
8. **if** $A_i = A_j$ & $B_i = B_j$ & $C_i \neq C_j$ **then** $R_i \neq R_j$

Given these rules, if 32 fragments were reduced in order such that they formed a 32-bit value, there only exists two solutions. Even without knowing the result of the first fragments from A, B, and C, the result must exist in

$\{0, 1\}$. Therefore, trying both options for the first fragment will give the results of the remaining fragments, forming 2×32-bit values.

Adding another obfuscated state for each fragment gives $fragments = \{1, 11\}$ and $states = \{0, 1, 2, 3\}$. Each fragment server now has three group combinations (mappings for each fragment to state), assuming two states per fragment, where $4!/(2!(4-2)!) = 6$, and only unique groups are needed $(6/2 = 3)$. When the fragment servers can see the obfuscated states using the reduction server model, this gives $3^3 = 27$ possible combinations, which reduces the problem back to two states, hence 27×2, giving 54 possibilities. Without the reduction server (using the performance model), because its own state is known to a fragment server, there would be $3^2 \times 2 = 18$ possibilities. Increasing the number of obfuscated states to four per fragment gives $fragments = \{1, 11\}$ and $states = \{0, 1, 2, 3, 4, 5, 6, 7\}$. The number of state mapping combinations increases to 35, where $8!/(4!(8-4)!) = 70$ and $70/2 = 35$. Both the reduction server and performance model have a much larger number of possibilities now, with $35^3 \times 2 = 85,750$ and $35^2 \times 2 = 2,450$ respectively. This was a simple example, where reducing a fragment which is not concatenated is pointless. However, it has shown how quickly the number of possibilities increases.

7.4.4 Many Fragment States

Looking at concatenated fragments, starting with two fragments being concatenated (an operation applied), gives $fragments = \{101, 1011, 1101, 11011\}$ and $states = \{0, 1, 2, 3\}$. The reduction server and performance model now have $(4!)^3 = 13,824$ and $(4!)^2 = 576$ possibilities for mappings respectively. The number of obfuscated states is doubled, where *states* is now $\{0, 1, 2, 3, 4, 5, 6, 7\}$, giving $8!/(2!(8-2)!) = 28$ grouping options. Therefore the number of mapping possibilities per fragment server is $8!/(2!(8-2)!) \times 6!/(2!(6-2)!) \times 4!/(2!(4-2)!) \times 2!/(2!(2-2)!) = 2520$. Overall possible mappings is now $(2520)^3 = 16,003,008,000$ and $(2520)^2 = 6,350,400$. The number of obfuscated states is again doubled, such that $states = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$, where there are now $16!/(4!(16-4)!) \times 12!/(4!(12-4)!) \times 8!/(4!(8-4)!) \times 4!/(4!(4-4)!) = 63,063,000$ possible mappings per fragment server, giving $63,063,000^3 \approx 2.5 \times 10^{23}$ and $63,063,000^2 \approx 3.9 \times 10^{15}$.

Allowing 4 concatenated states, where $fragments = \{1010101,\ 10101011,\ 10101101, 101011011,\ 10110101,\ 101101011, 101101101, 1011011011, 11010101,\ 110101011,\ 110101101,\ 1101011011,\ 110110101,\ 1101101011,\ 1101101101,\ 11011011011\}$, and gives 16 obfuscated states. Therefore, the number of possible options for the system is $(16!)^3 \approx 9.16 \times 10^{39}$ and $(16!)^2 \approx 4.38 \times 10^{26}$ for the reduction server or the performance model. Allowing two more fragments to be concatenated gives $(2^6!)^2 \approx 1.61 \times 10^{178}$

The number of possibilities given are purely theoretical and the best-case. But they show the large number of possible mapping options across a system using three fragment servers either using a reduction server, or the performance model. The attack vector of being able to guess the mapping options of the system has been mitigated with the enhanced privacy model, as no fragment server knows the exact obfuscated state of any other (can only see a vector of possible obfuscated states). The challenge with this attack is knowing if the current mapping for each fragment server has been found, whereas with encryption, it is mathematically provable that the correct decryption key was found.

7.4.5 Remarks on the Privacy Provided by FRIBs

This section has covered the security analysis for FRIBs, primarily the enhanced privacy model, as it offers the best privacy out of all the models presented in this thesis. Bin Encoding presented in Chapter 4 introduced the idea of Privacy-Preserving Encoding, where having knowledge of (1) the encoded data, (2) the algorithm for encoding and (3) the random mappings cannot directly decode the data. The protection/strength comes from the number of possible combinations and the difficulty of knowing whether possible decoded values are correct. The same is true with FRIBs, especially with the enhanced privacy model and one set of fragments. Naturally, if all the fragments are retrieved (leaked by a malicious entity), the only challenge is joining them together in the correct order. For example, fragments f_0, f_1, f_2, f_3 could be ordered f_2, f_1, f_3, f_0 on one fragment server, and f_2, f_3, f_0, f_1 on another. In this case, the program would need to know the ordering; however, it means that if the data is stolen, the fragments cannot be directly joined together.

There will exist other techniques to reduce the number of combinations,

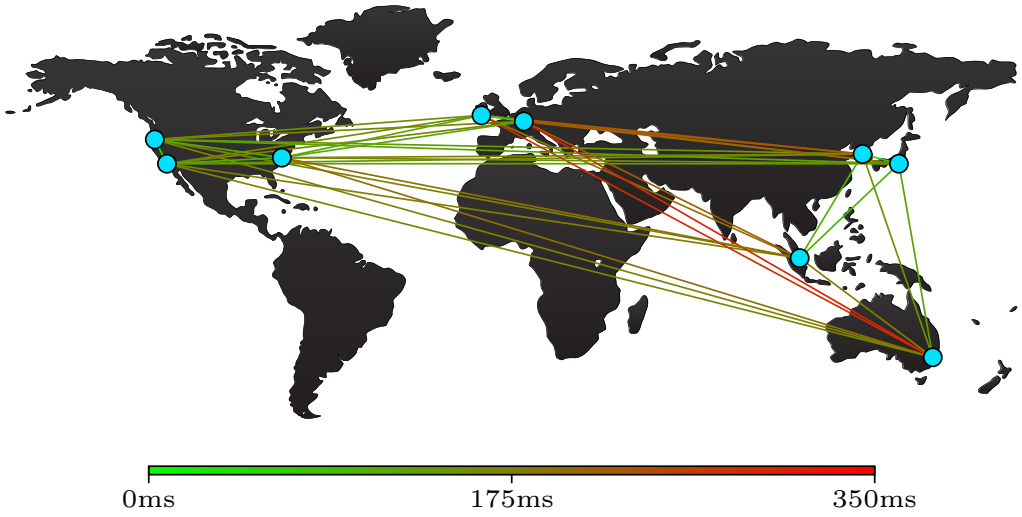

Figure 7.20: Network latency experienced across nine AWS datacenter locations

but these will be more implementation specific. Therefore, it is important to test any implementation to see if the states are being reached in a seemingly random manner. The security results presented for the voting implementation showed with an equal probability of a fragment server receiving a 0 or 1 and with the client randomly generating its fragment, there were no patterns in states or transitions. Therefore, if the fragment servers are hosted by different providers (AWS, Azure, Google Cloud Platform are a few examples) then a company's or personal data is hidden. Similarly, in the case of voting, if all fragment servers are hosted by different providers and managed by different organisations/entities (government, universities, companies to name a few), then a user's vote is hidden.

7.5 Other Areas of Analysis

A few other topics for analysis will be covered in this section: fragment server locations and thread mismatches. Another aspect of FRIBs is introduced–data privacy laws–which also play a role in keeping the fragments protected.

7.5.1 Latency

Figure 7.20 represents the average latency between nine AWS locations across the world. Note that all the AWS instances were free-tier so have the least priority for network and bandwidth. This figure shows that close groups have RTT ranging between $20 - 100$ms for the majority, whereas locations such as Sydney do not have usable RTT times. Section 7.2.4 showed that parallelisation can hide latency; however, with large physical distances (cross-continent), then the traffic is susceptible to higher packet loss and reduced bandwidth. Therefore, for high-performance implementations of FRIBs, having all the fragment servers in the same/near countries with high bandwidth is required.

7.5.2 Data Privacy Laws

Laws governing data privacy differ between countries and jurisdictions [174]. This means organisations and personal users need to understand where their data is physically at rest, not just which country the user resides in. Given that FRIBs has been designed to protect against any single entity, if a law enforcement agency thinks some data/user is malicious or is under investigation, their data can be requested. However, the requests will be to each different hosting company, which can be in different jurisdictions, thus differently the laws that cover the data. For example, if fragments are hosted in country A and country B, the strength and/or weaknesses of both protect the privacy of the data. If country A has weak privacy laws, but country B has strict laws, then even if the fragments from country A are requested and received, the legal process of country B still needs to be followed. If all organisations/countries agree that the data should be handed over to law enforcement, then the data can be decoded, but if only one country or state thinks the data should be handed over, then the data is still protected. Some examples of country and company data policies are given in Appendix E, to help highlight how distributing data can improve data protection and spread the decision for data requests across different courts and judges. This thesis does not claim the novelty of distributing data for distributing data privacy laws, but instead shows it is another property offered by FRIBs.

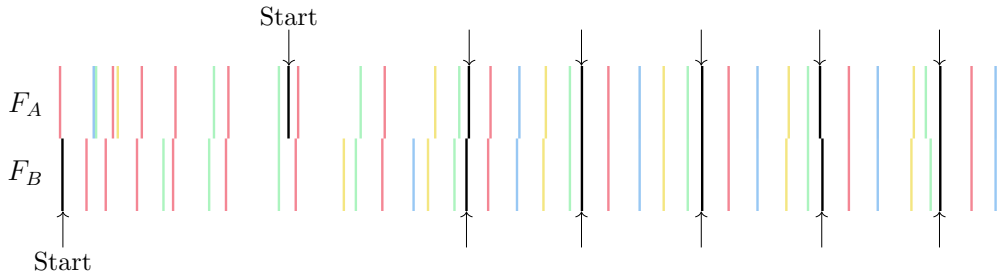

Figure 7.21: Thread scheduling across two fragment servers for fifty randomly started user jobs

7.5.3 Threading Mismatches

Each fragment server executes the same instructions before a reduction is needed, and all the fragments need to be received before a reduction can return a result. Ideally, all the servers are running the user's threads at the same time; in reality, this cannot be guaranteed. A test was executed where 50 user threads were started in random order with a 10ms delay between each, where the thread order for two fragment servers was recorded. The user task computes 100 addition operations using the performance model implementation in Lisp. Figure 7.21 shows 10% of the threads for the first few addition operations, where each line represents the start of an addition operation across both servers, and the colour represents a unique user task. The labelled thread starts early on F_B, but takes longer to start on F_A. However, the start of the next five additions happens at nearly the same time on both 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.

7.5.4 Pseudo-Random Number Generators

The security strength of pseudo-random number generators is out-of-scope of this thesis; however, to mitigate the chance or consequences of the target fragment server learning the seed used to obfuscated its result LUT, we can periodicity change the seed. One of the other two fragment servers can use a true random number generator to create a new seed, and securely send it to the remaining fragment server, or both could work together to create a new seed–

distributing the entropy–but this would require more network communication.

A commonly used pseudo-random number generator (including Python) is the Mersenne Twister [175], which due to linear recursion is not considered cryptographically secure [176]. The output of the pseudo-random number generator is hidden from the target fragment server because it is XORed with the LUT row, making it difficult to be able to recover the output needed to predict new output. Other techniques can make the Mersenne Twister secure, including using a secure hashing algorithm or combining multiple outputs. Therefore, even though analysing the security of pseudo-random number generators is out-of-scope, the combination of periodicity changing seeds, the output of the pseudo-random number generator being XORed, and the fact pseudo-random number generators can be cryptographically secure, means the security strength of FRIBs should not be compromised by the use of pseudo-random number generators.

7.6 Summary

This chapter demonstrated implementations for both cloud voting and searching, while describing the wrapper functions and processes (scheduler and LUT generation). The primary hypothesis (Hypothesis 3) of this thesis is to find a method where privacy-preserving processing in the cloud can be practical. In terms of the definition of practicality in Chapter 3, performance is the main attribute being evaluated for computation on the client and within the cloud. For client performance, FRIBs will be similar to plain text except for the fragmentation generation and that data needs to be sent to multiple servers. Partially homomorphic encryption schemes with the required zero-knowledge proof cannot match the client-side performance of FRIBs, especially within web browsers. FRIBs also offers reasonable computation performance in the cloud for dedicated implementations, where it could process 174 votes per second. Using one server, a partially homomorphic encryption scheme with zero-knowledge proofs could achieve a maximum of 91 votes per second, and with the same number of servers as FRIBs, it could reach 273 votes per second. Even though the cloud performance of FRIBs is not at the same level as partially homomorphic encryption, taking into consideration client-side perfor-

mance, the two techniques have their claims for having the best performance.

The second aspect of practicality is utility and flexibility, for both the client and cloud. Partially homomorphic encryption can only add a single vote to the tally, where FRIBs can implement more advanced voting schemes. So, in terms of utility and performance, FRIBs is more practical than existing partially homomorphic encryption schemes. This chapter also presented a searching example using FRIBs, where the lower layer implementation was generic, so any program could be built upon it without regenerating new result LUTs. This is a feature that traditional multi-party computation techniques lack, along with LUT reuse. However, the fact that FRIBs only reveals the servers next state, and hides some of the inputs to the lookup, means it can reuse the same LUTs. This was proven in Section 7.4, where there was no bias between states reached and transitions between states, while the vector received had an equal probability of containing any state. This means FRIBs is able to offer privacy-preserving processing without the need for the data to be encrypted by the client or on the fragment servers, answering Hypothesis 3 (data is only encrypted when transmitted on the network).

8

Conclusions and Future Work

8.1 Revisiting the Hypotheses

As discussed, Hypotheses 1 and 2 did not lead to any breakthroughs for answering the research question "Can fully homomorphic encryption be practical in the cloud?". This question still remains open, as Hypothesis 3 widened the scope from only homomorphic encryption to any privacy-preserving technique. Hypothesis 3 was realised with the novel scheme presented in this thesis, Fragmenting Individual Bits (FRIBs), a form of multi-party computation that uses encoding and distribution to protect privacy. Chapter 5 introduced the concept of FRIBs, presenting two core models: a performance model and an enhanced privacy model. The enhanced privacy model requires twice the network latency as the performance mode, but offers greater privacy and therefore is the recommended model. The models use LUTs to compute operations, and support generic logic (NAND gates) and custom functions, as discussed in Chapter 6. Each result from the LUT only gives the new fragment value (a single bit) for one fragment server; therefore, keeping LUT sizes small. The LUTs for the enhanced privacy model can also be reused without revealing patterns, addressing an issue for most multi-party computation schemes. Other improvements over current multi-party computation schemes are that homomorphic encryption is not required and less network data is transmitted.

Two implementations are given in Chapter 7, representing the different aspects of practicality in order to prove that FRIBs can be practical in the cloud. The voting implementation allowed the comparing of FRIBs to the schemes in

Chapter 3, showing that the performance is close to that of partially homomorphic encryption in the cloud. Client-side performance is actually comparable to plain text, which was a limitation of partially homomorphic encryption, especially for the survey application in Chapter 3. The second implementation was an exact searching function, which can be combined with the work in Chapter 4–Bin Encoding–for faster search times. This work showed the flexibility of FRIBs and how programs could be written offline before automatically running on top of FRIBs in the cloud.

The research presented in this thesis has endeavoured to find the balance between security, performance and utility. Both performance and utility were met by FRIBs, but Hypothesis 3 also required privacy to be protected (the security aspect). Chapter 7 covered the privacy analysis for FRIBs, where the enhanced privacy model gave acceptable results; the value cannot be decoded with one set of fragments even while spying on the server during operation. The experiments included analysing the patterns in the states received in the request vector and patterns in the fragment servers own states. For the voting implementation (the primary use case), there was an equal probability of any fragment server state being reached, even with millions of *no* votes cast. With an equal probability of reaching any state, and an equal probability of receiving a 1 or 0 fragment (even with all *no* votes), means FRIBs provides sufficient voter privacy. The values can only be decoded if all the fragments are retrieved (stolen or leaked) similar to threshold cryptography; therefore, encoding and distribution can protect data privacy while being practical, where basic encryption is only needed for network transfers, thus realising Hypothesis 3.

8.2 Revisiting the Use Cases

An implementation for the first use case given in Section 1.3.1 for voting has been presented in Section 7.2. A vote of 0 or 1 can be fragmented, sent to each fragment server, verified, then added to the tally of the ballot. During the fragmentation, because the vote is a single bit, no proofs are required by the user to prove the vote is valid–an improvement over partially homomorphic encryption. Restrictions such as only one vote per set of candidates can be computed by the fragment servers without revealing any information about the

vote. Votes are added to the tally, allowing millions of users to cast electronic votes while protecting their privacy.

Low powered devices are collecting and sending data to the cloud for analysis in the second use case. Chapter 3 showed that client performance is also critical for a practical scheme, where encrypting and generating proofs could take minutes on a mobile device. With the simple fragmentation algorithm for FRIBs requiring very little computational power, a device can use already optimised encryption algorithms [115] to send the fragments to the fragment servers; giving FRIBs superior client side performance over other solutions. The data collected by the low powered device is kept privately in the cloud at all times while maintaining its functionality. The second use case mentions heartbeat monitoring, which could be analysed using neural networks [177][178]. Once trained, neural networks only use basic functions which could then identify irregularities in health data. The fragment servers need to be split across organisations, where the models (or functions) can be kept confidential from the user, while the users' data is kept private from the organisations.

The third use case has companies moving to cloud environments, but requires data to remain secure, even from the cloud hosting companies. With this requirement, encryption either needs to be performed on a client device or a company gateway before being transmitted to the cloud. Fully homomorphic encryption cannot be used due to processing overheads; therefore, any processing requires downloading the data from the cloud back to a device. A multi-party computation scheme such as FRIBs would give more flexibility than current homomorphic encryption schemes. FRIBs allows LUTs to be reused for different users and applications within the company, allowing a more user-centric implementation over other multi-party computation schemes.

Sharing data between organisations in the cloud is the final use case in Section 1.3.1. For example, with the health sector, patient records need to remain private, thus making data sharing difficult. Data anonymisation techniques exist [144][145][146] but are not perfect [179], where the FRIBs scheme does not need data to be anonymised. Each organisation can host a fragment server, and fragmented patient data can be stored across all the fragment servers (all organisations); therefore, each organisation has fragments of all the data but cannot do anything with them. When all the fragment servers perform the

same operation, only then can operations be performed over the data; if an organisation does not agree with the function, they can deny it and the function cannot be run. When data is shared using an anonymisation technique, it is difficult to recall the data – control is no longer with the owner of the data–whereas with FRIBs, the organisation can delete the fragments on its server, rendering the other fragments useless.

Use cases presented in Section 1.3.1 are implementable with FRIBs, allowing for practical privacy-preserving processing in the cloud to protect users and organisations with current technologies. Not all functions or applications would be possible, but FRIBs provides more flexibility over homomorphic encryption and other multi-party computation schemes.

8.3 Future Work

The research presented in this thesis has proven the plausibility of using encoding for privacy-preserving processing in the cloud. However, there are a number of areas that can be explored for the FRIBs scheme.

The functionality to hide the program being executed from the cloud exists but has not been explored. Custom LUTs can be used, and instead of the program having the instruction *ADD*, it could be *FUN1*. The reason hiding programs has not been tested is because the patterns of each function could reveal the operation, where an add or multiply will have a different number of reductions (even if named *FUN1* and *FUN2*). Combining the hidden function name with the redundancy model would help hide the program even further if the redundancy aspect is not actually required, because then purposeful corruption could be applied to each instruction set received by the fragments servers, where one always contains an error.

A useful property for FRIBs would be the ability to bootstrap, where it can generate new LUTs in the cloud without revealing any information or needing a client. Because data is fragmented using a single algorithm, the data does not need to be re-encrypted to use any new LUT. This was not researched, because it is very easy for a client to generate the LUTs and securely send them to the fragment servers.

Currently, there is no verification or proof mechanism built into FRIBs where

the fragment servers could prove the operation and the bit fragments operated on. There is a form of verification for operations if there is at least one honest fragment server, as the other fragment servers cannot run arbitrary functions. Hash chaining could be used to provide a form of accounting; for example, when a fragment server receives a fragment, it could also receive the hash values for other fragment servers. During execution, each fragment server can chain the hashes for the corresponding fragments. All the hash chains should give the same value across each fragment server.

System level mechanisms can also be put in place to harden security. A dedicated server could be built to only run the FRIBs scheme, for example, using an FPGA [36] or programming on bare metal. In a virtual environment, while an operation is being performed, central processing unit task switching could be disabled, keeping states in registers, and can be encrypted before leaving the central processing unit. Given all the fragment servers have to process fragments at the same time, storage locks could be designed so that each fragment server needs to approve the access to the fragment.

Further analysis is required for the network traffic produced by FRIBs compared to other multi-party computation schemes, especially in terms of the offline processing (for example, generating the triples [95]). There are possible improvements for performing operations faster, such as multiplication, to get it closer to the performance of addition with pipelining. Finally, getting a real-world system to use FRIBs and to promote further development on the scheme would be a big goal for the future.

8.4 Final Remarks

The concept of using encoding instead of encryption for privacy-preserving processing is possible with FRIBs; however, encryption still remains important for network transfers, otherwise all the fragments could be stolen when transferred to the fragment servers. Another important requirement is the fragment servers should be hosted in different cloud environments, both virtual and physical, to prevent any single entity having access to more than one fragment server. Aside from that, FRIBs offers a balance between security, performance and utility for cloud computing, unlike state-of-the-art techniques presented in

Chapter 2. FRIBs is a subset of multi-party computation but does not require the garbled circuits or inputs to be encrypted, while also allowing the LUTs (garbled circuits or garbled gates) to be reused. In terms of multi-party computation versus homomorphic encryption, both have positives; homomorphic encryption ultimately has better security, but distributing data such that it cannot be reconstructed without each fragment also distributes trust, privacy laws, system level security, and allows for greater redundancy support.

References

[1] Ryan KL Ko, Giovanni Russello, Richard Nelson, Shaoning Pang, Aloysius Cheang, Gill Dobbie, Abdolhossein Sarrafzadeh, Sivadon Chaisiri, Muhammad Rizwan Asghar, and Geoffrey Holmes. STRATUS: Towards Returning Data Control to Cloud Users. In *International Conference on Algorithms and Architectures for Parallel Processing*, pages 57–70. Springer, 2015.

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

[3] Fujitsu Limited. Personal Data in the Cloud: A Global Survey of Consumer Attitudes. Online. Retrieved from http://www.fujitsu.com/downloads/SOL/fai/reports/fujitsu_personal-data-in-the-cloud.pdf [Accessed 22/03/17], 2010.

[4] Ronald L. Rivest, Len Adleman, and Michael L. Dertouzos. On Data Banks and Privacy Homomorphisms. *Foundation of Secure Computation*, Academic Press, New York:169–179, 1978.

[5] Taher ElGamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *Advances in Cryptology*, pages 10–18. Springer, 1985.

[6] Pascal Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In *Advances in cryptology—EUROCRYPT'99*, pages 223–238. Springer, 1999.

[7] Craig Gentry. Fully Homomorphic Encryption using Ideal Lattices. In *Proceedings of the 41st annual ACM symposium on Symposium on theory of computing-STOC\'09*, pages 169–169. ACM Press, 2009.

[8] Marten Van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully Homomorphic Encryption over the Integers. In *Advances in Cryptology–EUROCRYPT 2010*, pages 24–43. Springer, 2010.

[9] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *Advances in Cryptology–CRYPTO 2013*, pages 75–92. Springer, 2013.

[10] Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang, and Berk Sunar. Exploring the Feasibility of Fully Homomorphic Encryption. *Computers, IEEE Transactions on*, 64(3):698–706, 2015.

[11] Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping Homomorphic Encryption in Less than a Second. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 617–640. Springer, 2015.

[12] Cynthia Dwork. Differential Privacy. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *Automata, Languages and Programming*, pages 1–12, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[13] Cynthia Dwork. Differential Privacy: A Survey of Results. In Manindra Agrawal, Dingzhu Du, Zhenhua Duan, and Angsheng Li, editors, *Theory and Applications of Models of Computation*, pages 1–19, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[14] Eyad Saleh, Ahmad Alsa'deh, Ahmad Kayed, and Christoph Meinel. Processing Over Encrypted Data: Between Theory and Practice. *ACM SIGMOD Record*, 45(3):5–16, 2016.

[15] Andrew C Yao. Protocols for Secure Computations. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 160–164. IEEE, 1982.

[16] Andrew Chi-Chih Yao. How to Generate and Exchange Secrets. In *Foundations of Computer Science, 1986., 27th Annual Symposium on*, pages 162–167. IEEE, 1986.

[17] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to Play any Mental Game. In *Proceedings of the nineteenth annual ACM symposium on Theory of Computing*, pages 218–229. ACM, 1987.

[18] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. NTRU: A Ring-Based Public Key Cryptosystem. In *International Algorithmic Number Theory Symposium*, pages 267–288. Springer, 1998.

[19] Damien Stehlé and Ron Steinfeld. Making NTRU as Secure as Worst-Case Problems over Ideal Lattices. In *Eurocrypt*, volume 6632, pages 27–47. Springer, 2011.

[20] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-Fly Multiparty Computation on the Cloud via Multikey Fully Homomorphic Encryption. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 1219–1234. ACM, 2012.

[21] G Edward Suh, Charles W O'Donnell, and Srinivas Devadas. AEGIS: A Single-Chip Secure Processor. *Information Security Technical Report*, 10(2):63–73, 2005.

[22] Mark A Will, Ryan KL Ko, and Ian H Witten. Bin Encoding: A User-Centric Secure Full-Text Searching Scheme for the Cloud. In *Trustcom/BigDataSE/ISPA, 2015 IEEE*, volume 1, pages 563–570. IEEE, 2015.

[23] Mark A Will, Ryan KL Ko, and Ian H Witten. Privacy Preserving Computation by Fragmenting Individual Bits and Distributing Gates. In *The 15th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom-16)*, volume 1, pages 900–908. IEEE, 2016.

[24] Encoding vs. Encryption vs. Hashing vs. Obfuscation. Online. Retrieved from https://danielmiessler.com/study/encoding-encryption-hashing-obfuscation/ [Accessed 30/06/18].

[25] Auguste Kerckhoffs. *La cryptographie militaire*. Librairie militaire de L. Baudoin, 1883.

[26] Peter Gutmann. New Zealand Crypto Policy - Confusion now Hath Made His Masterpiece. Online. Retrieved from https://www.cs.auckland.ac.nz/~pgut001/policy/ [Accessed 03/07/17].

[27] John Borrie. MFAT Letter of 13 February 1997 to Peter Gutmann. Online. Retrieved from https://www.cs.auckland.ac.nz/~pgut001/policy/mfat_3r.html [Accessed 03/07/17].

[28] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A Survey of Mobile Malware in the Wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2011.

[29] Mark A Will, Brandon Nicholson, Marc Tiehuis, and Ryan KL Ko. Secure Voting in the Cloud using Homomorphic Encryption and Mobile Agents. In *2015 International Conference on Cloud Computing Research and Innovation (ICCCRI)*, pages 173–184. IEEE, 2015.

[30] FireEye. ZERO-DAY DANGER: A Survey of Zero-Day Attacks and What They Say About the Traditional Security Model. White Paper, 2015.

[31] Internet Security Threat Report. Technical report, Symantec, Volume 21, Online. Retrieved from https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf [Accessed 30/06/18], April, 2016.

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

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

[34] Ronald Rivest, Adi Shamir, and Leonard Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[35] Andrew C Yao. Theory and Application of Trapdoor Functions. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on*, pages 80–91. IEEE, 1982.

[36] Mark A. Will and Ryan K. L. Ko. Secure FPGA as a Service — Towards Secure Data Processing by Physicalizing the Cloud. *IEEE Trustcom/BigDataSE/ICESS*, pages 449–455, 08 2017.

[37] Mark A. Will and Ryan K. L. Ko. *The Cloud Security Ecosystem, 1st Edition, Technical, Legal, Business and Management Issues*, chapter 5 – A Guide to Homomorphic Encryption. Syngress, an Imprint of Elsevier, 2015.

[38] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-DNF Formulas on Ciphertexts. In Joe Kilian, editor, *Theory of Cryptography*, pages 325–341, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[39] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford University, 2009.

[40] Frederik Armknecht and Ahmad-Reza Sadeghi. A New Approach for Algebraically Homomorphic Encryption. *IACR Cryptology ePrint Archive*, 2008:422, 2008.

[41] Josep Domingo-Ferrer. A Provably Secure Additive and Multiplicative Privacy Homomorphism. In *Information Security*, pages 471–483. Springer, 2002.

[42] Josep Domingo i Ferrer. A New Privacy Homomorphism and Applications. *Information Processing Letters*, 60(5):277–282, 1996.

[43] Ernest F Brickell and Yacov Yacobi. On Privacy Homomorphisms. In *Advances in Cryptology—EUROCRYPT'87*, pages 117–125. Springer, 1988.

[44] Niv Ahituv, Yeheskel Lapid, and Seev Neumann. Processing Encrypted Data. *Communications of the ACM*, 30(9):777–780, 1987.

[45] GR Blakley and Catherine Meadows. A Database Encryption Scheme which Allows the Computation of Statistics using Encrypted Data. In

Security and Privacy, 1985 IEEE Symposium on, pages 116–116. IEEE, 1985.

[46] Dorothy E Denning. Signature Protocols for RSA and Other Public-Key Cryptosystems. 1982.

[47] Shafi Goldwasser and Silvio Micali. Probabilistic Encryption & How To Play Mental Poker Keeping Secret All Partial Information. In *Proceedings of the fourteenth annual ACM symposium on Theory of Computing*, pages 365–377. ACM, 1982.

[48] George I Davida, David L Wells, and John B Kam. A Database Encryption System with Subkeys. *ACM Transactions on Database Systems (TODS)*, 6(2):312–328, 1981.

[49] David Wagner. Cryptanalysis of an Algebraic Privacy Homomorphism. In *Information Security*, pages 234–239. Springer, 2003.

[50] Miklós Ajtai. Generating Hard Instances of Lattice Problems. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 99–108. ACM, 1996.

[51] Daniele Micciancio and Shafi Goldwasser. *Complexity of Lattice Problems: A Cryptographic Perspective*, volume 671. Springer, 2002.

[52] Oded Goldreich, Daniele Micciancio, Shmuel Safra, and J-P Seifert. Approximating Shortest Lattice Vectors is not Harder than Approximating Closest Lattice Vectors. *Information Processing Letters*, 71(2):55–61, 1999.

[53] Peter van Emde-Boas. *Another NP-Complete Partition Problem and the Complexity of Computing Short Vectors in a Lattice*. Department, Univ., 1981.

[54] Daniele Micciancio. The Shortest Vector in a Lattice is Hard to Approximate to Within some Constant. *SIAM journal on Computing*, 30(6):2008–2035, 2001.

[55] O Regev. The Learning with Errors Problem (Invited Survey). In *Computational Complexity (CCC), 2010 IEEE 25th Annual Conference on*, pages 191–204. IEEE, 2010.

[56] Chris Peikert. Public-Key Cryptosystems from the Worst-Case Shortest Vector Problem. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 333–342. ACM, 2009.

[57] Daniel J Bernstein. Introduction to Post-Quantum Cryptography. In *Post-quantum cryptography*, pages 1–14. Springer, 2009.

[58] Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-Tolerant Learning, The Parity Problem, and the Statistical Query Model. *Journal of the ACM (JACM)*, 50(4):506–519, 2003.

[59] Miklós Ajtai, Ravi Kumar, and Dandapani Sivakumar. A Sieve Algorithm for the Shortest Lattice Vector Problem. In *Proceedings of the thirty-third annual ACM symposium on Theory of computing*, pages 601–610. ACM, 2001.

[60] Daniele Micciancio and Panagiotis Voulgaris. A Deterministic Single Exponential Time Algorithm for most Lattice Problems based on Voronoi Cell Computations. *SIAM Journal on Computing*, 42(3):1364–1391, 2013.

[61] Marvin Marcus. *Introduction to Linear Algebra*. Courier Dover Publications, 1988.

[62] Craig Gentry, Shai Halevi, and Nigel P. Smart. *Homomorphic Evaluation of the AES Circuit*, pages 850–867. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[63] Craig Gentry and Shai Halevi. Implementing Gentry's Fully-Homomorphic Encryption Scheme. In *Advances in Cryptology–EUROCRYPT 2011*, pages 129–148. Springer, 2011.

[64] Jacob Alperin-Sheriff and Chris Peikert. Faster Bootstrapping with Polynomial Error. In *International Cryptology Conference*, pages 297–314. Springer, 2014.

[65] Chris Peikert et al. A Decade of Lattice Cryptography. *Foundations and Trends® in Theoretical Computer Science*, 10(4):283–424, 2016.

[66] Zvi Galil, Stuart Haber, and Moti Yung. *Cryptographic Computation: Secure Fault-Tolerant Protocols and the Public-Key Model (Extended Abstract)*, pages 135–155. Springer Berlin Heidelberg, Berlin, Heidelberg, 1988.

[67] David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty Unconditionally Secure Protocols. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 11–19. ACM, 1988.

[68] T. Rabin and M. Ben-Or. Verifiable Secret Sharing and Multiparty Protocols with Honest Majority. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC '89, pages 73–85, New York, NY, USA, 1989. ACM.

[69] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. *Secure Two-Party Computation Is Practical*, pages 250–267. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[70] Jesper Buus Nielsen, Peter Sebastian Nordholt, Claudio Orlandi, and Sai Sheshank Burra. *A New Approach to Practical Active-Secure Two-Party Computation*, pages 681–700. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[71] Niklas Buescher and Stefan Katzenbeisser. Faster Secure Computation through Automatic Parallelization. In *USENIX Security*, pages 531–546, 2015.

[72] Jesper Buus Nielsen and Claudio Orlandi. LEGO for Two-Party Secure Computation. In *TCC*, volume 5444, pages 368–386. Springer, 2009.

[73] Tore Kasper Frederiksen, Thomas Pelle Jakobsen, Jesper Buus Nielsen, Peter Sebastian Nordholt, and Claudio Orlandi. Minilego: Efficient secure two-party computation from general assumptions. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 537–556. Springer, 2013.

[74] Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. Tinygarble: Highly compressed and scalable sequential garbled circuits. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 411–428. IEEE, 2015.

[75] Yongge Wang, Qutaibah M. Malluhi, and Khaled MD Khan. Garbled Computation in Cloud. *Future Generation Computer Systems*, 62:54 – 65, 2016.

[76] Ronald Cramer, Ivan Damgård, and Jesper Nielsen. Multiparty Computation from Threshold Homomorphic Encryption. *Advances in cryptology—EUROCRYPT 2001*, pages 280–300, 2001.

[77] Ronald Cramer, Ivan B Damgård, Stefan Dziembowski, Martin Hirt, and Tal Rabin. Efficient Multiparty Computations with Dishonest Minority. *BRICS Report Series*, 5(36), 1998.

[78] Matthew Franklin and Moti Yung. Communication Complexity of Secure Computation (Extended Abstract). In *Proceedings of the Twenty-fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 699–710, New York, NY, USA, 1992. ACM.

[79] Shafi Goldwasser, Yael Kalai, Raluca Ada Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. Reusable Garbled Circuits and Succinct Functional Encryption. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 555–564. ACM, 2013.

[80] Benjamin Mood, Debayan Gupta, Kevin Butler, and Joan Feigenbaum. Reuse It or Lose It: More Efficient Secure Computation through Reuse of Encrypted Values. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 582–596. ACM, 2014.

[81] Xu An Wang, Fatos Xhafa, Jianfeng Ma, Yunfei Cao, and Dianhua Tang. Reusable Garbled Gates for New Fully Homomorphic Encryption Service. *International Journal of Web and Grid Services*, 13(1):25–48, 2017.

[82] Amos Beimel. Secret-Sharing Schemes: A Survey. In *International Conference on Coding and Cryptology*, pages 11–46. Springer, 2011.

[83] George Robert Blakley. Safeguarding cryptographic keys. *Proceedings of Americian Federation of Information Processing Societies (AFIPS'79) National Computer Conference*, pages 313–317, 1979.

[84] Adi Shamir. How to Share a Secret. *Communications of the ACM*, 22(11):612–613, November 1979.

[85] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. Asynchronous multiparty computation: Theory and implementation. In *International Workshop on Public Key Cryptography*, pages 160–179. Springer, 2009.

[86] VIFF, the Virtual Ideal Functionality Framework. Online. Retrieved from http://viff.dk [Accessed 06/06/17].

[87] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-preserving Aggregation of Multi-domain Network Events and Statistics. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 223–240, Berkeley, CA, USA, 2010. USENIX Association.

[88] Martin Burkhart. SEPIA. Online. Retrieved from http://sepia.ee.ethz.ch/index.html [Accessed 08/06/17].

[89] Dahlia Malkhi, Noam Nisan, Benny Pinkas, Yaron Sella, et al. Fairplay-Secure Two-Party Computation System. In *USENIX Security Symposium*, volume 4. San Diego, CA, USA, 2004.

[90] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: A System for Secure Multi-Party Computation. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 257–266. ACM, 2008.

[91] Fair Play. Online. Retrieved from https://www.cs.huji.ac.il/project/Fairplay/home.html [Accessed 06/06/17].

[92] A Aly, M Keller, E Orsini, D Rotaru, P Scholl, N Smart, and T Wood. SCALE and MAMBA Documentation. On-

line. Retrieved from https://homes.esat.kuleuven.be/~nsmart/SCALE/Documentation.pdf [Accessed 29/01/19], 2018.

[93] I. Damgard, V. Pastro, N.P. Smart, and S. Zakarias. Multiparty Computation from Somewhat Homomorphic Encryption. Cryptology ePrint Archive, Report 2011/535, 2011.

[94] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P Smart. Practical Covertly Secure MPC for Dishonest Majority–or: Breaking the SPDZ Limits. In *European Symposium on Research in Computer Security*, pages 1–18. Springer, 2013.

[95] Donald Beaver. Efficient Multiparty Protocols using Circuit Randomization. In *Annual International Cryptology Conference*, pages 420–432. Springer, 1991.

[96] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *European Symposium on Research in Computer Security*, pages 192–206. Springer, 2008.

[97] Cybernetica. Sharemind. Online. Retrieved from https://sharemind.cyber.ee [Accessed 06/06/17].

[98] Liina Kamm. Using Sharemind to Estimate Satellite Collision Probability. Online. Retrieved from https://sharemind.cyber.ee/satellite-collision-security/ [Accessed 06/06/17], April 2015.

[99] Azer Bestavros Frederick Jansen Mayank Varia Andrei Lapets, Nikolaj Volgushev. Secure Multi-Party Computation for Analytics Deployed as a Lightweight Web Application. Technical report, CS Dept, Boston University, July 2016.

[100] Andrei Lapets, Nikolaj Volgushev, Azer Bestavros, Frederick Jansen, and Mayank Varia. Secure MPC for Analytics as a Web Application. In *Cybersecurity Development (SecDev), IEEE*, pages 73–74. IEEE, 2016.

[101] Azer Bestavros, Andrei Lapets, and Mayank Varia. User-Centric Distributed Solutions for Privacy-Preserving Analytics. *Communications of the ACM*, 60(2):37–39, 2017.

[102] Bo Yang, Kaijie Wu, and Ramesh Karri. Scan Based Side Channel Attack on Dedicated Hardware Implementations of Data Encryption Standard. In *International Test Conference, 2004. Proceedings. ITC 2004.*, pages 339–344. IEEE, 2004.

[103] Boris Köpf and David Basin. An Information-Theoretic Model for Adaptive Side-Channel Attacks. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 286–296. ACM, 2007.

[104] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Advances in Cryptology—CRYPTO'99*, pages 388–397. Springer, 1999.

[105] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic Analysis: Concrete Results. In *Cryptographic Hardware and Embedded Systems—CHES 2001*, pages 251–261. Springer, 2001.

[106] Amazon EC2 F1 Instances: Run Customizable FPGAs in the AWS Cloud. Online. Retrieved from https://aws.amazon.com/ec2/instance-types/f1/ [Accessed 19/06/17].

[107] E.M. Songhori, S. Zeitouni, G. Dessouky, T. Schneider, A.-R. Sadeghi, and F. Koushanfar. GarbledCPU: A MIPS Processor for Secure Computation in Hardware. In *Proceedings - Design Automation Conference*, 2016.

[108] Blaise Gassend, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. Silicon Physical Random Functions. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 148–160. ACM, 2002.

[109] G Edward Suh and Srinivas Devadas. Physical Unclonable Functions for Device Authentication and Secret Key Generation. In *Proceedings of the 44th annual Design Automation Conference*, pages 9–14. ACM, 2007.

[110] Meng-Day Mandel Yu and Srinivas Devadas. Secure and Robust Error Correction for Physical Unclonable Functions. *IEEE Design & Test of Computers*, 27(1):48–65, 2010.

[111] Jorge Guajardo, Sandeep S Kumar, Geert-Jan Schrijen, and Pim Tuyls. Physical Unclonable Functions and Public-Key Crypto for FPGA IP Protection. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 189–195. IEEE, 2007.

[112] Abhranil Maiti, Logan McDougall, and Patrick Schaumont. The Impact of Aging on an FPGA-based Physical Unclonable Function. In *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, pages 151–156. IEEE, 2011.

[113] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES-The Advanced Encryption Standard*. Springer Science & Business Media, 2013.

[114] Alireza Hodjat and Ingrid Verbauwhede. A 21.54 Gbits/s Fully Pipelined AES Processor on FPGA. In *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on*, pages 308–309. IEEE, 2004.

[115] Gaël Rouvroy, F-X Standaert, J-J Quisquater, and J-D Legat. Compact and Efficient Encryption/Decryption Module for FPGA Implementation of the AES Rijndael very Well Suited for Small Embedded Applications. In *Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004. International Conference on*, volume 2, pages 583–587. IEEE, 2004.

[116] Tim Good and Mohammed Benaissa. AES on FPGA from the Fastest to the Smallest. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 427–440. Springer, 2005.

[117] Mohamed Khalil Hani, Tan Siang Lin, and Nasir Shaikh-Husin. FPGA Implementation of RSA Public-key Cryptographic Coprocessor. In *TENCON 2000. Proceedings*, volume 3, pages 6–11. IEEE, 2000.

[118] Alan Daly and William Marnane. Efficient Architectures for Implementing Montgomery Modular Multiplication and RSA Modular Exponentiation on Reconfigurable Logic. In *Proceedings of the 2002 ACM/SIGDA*

tenth international symposium on Field-programmable gate arrays, pages 40–49. ACM, 2002.

[119] Alessandro Cilardo, Antonino Mazzeo, Luigi Romano, and Giacinto Paolo Saggese. Exploring the Design-Space for FPGA-based Implementation of RSA. *Microprocessors and Microsystems*, 28(4):183–191, 2004.

[120] Darrel Hankerson, Alfred J Menezes, and Scott Vanstone. *Guide to Elliptic Curve Cryptography*. Springer Science & Business Media, 2006.

[121] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 119–132. Springer, 2004.

[122] Kristin Lauter. The Advantages of Elliptic Curve Cryptography for Wireless Security. *IEEE Wireless communications*, 11(1):62–67, 2004.

[123] Ciaran J McIvor, Maire McLoone, and John V McCanny. Hardware Elliptic Curve Cryptographic Processor Over $rmGF(p)$. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 53(9):1946–1957, 2006.

[124] KH Leung, KW Ma, Wai Keung Wong, and Philip Heng Wai Leong. FPGA Implementation of a Microcoded Elliptic Curve Cryptographic Processor. In *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, pages 68–76. IEEE, 2000.

[125] Guerric Meurice de Dormale and Jean-Jacques Quisquater. High-Speed Hardware Implementations of Elliptic Curve Cryptography: A Survey. *Journal of systems architecture*, 53(2):72–84, 2007.

[126] William N Chelton and Mohammed Benaissa. Fast Elliptic Curve Cryptography on FPGA. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 16(2):198–205, 2008.

[127] Thomas Wollinger and Christof Paar. *How Secure Are FPGAs in Cryptographic Applications?*, pages 91–100. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[128] Dan Boneh, Giovanni Di Crescenzo, Rafail Ostrovsky, and Giuseppe Persiano. Public Key Encryption with Keyword Search. In *Advances in Cryptology-Eurocrypt 2004*, pages 506–522. Springer, 2004.

[129] Yong Ho Hwang and Pil Joong Lee. Public Key Encryption with Conjunctive Keyword Search and its Extension to a Multi-User System. In *Pairing-Based Cryptography–Pairing 2007*, pages 2–22. Springer, 2007.

[130] Joonsang Baek, Reihaneh Safavi-Naini, and Willy Susilo. Public Key Encryption with Keyword Search Revisited. In *Computational Science and Its Applications–ICCSA 2008*, pages 1249–1259. Springer, 2008.

[131] Jin Li, Qian Wang, Cong Wang, Ning Cao, Kui Ren, and Wenjing Lou. Fuzzy Keyword Search over Encrypted Data in Cloud Computing. In *Proceedings of the 29th Conference on Information Communications*, pages 441–445. IEEE Press, 2010.

[132] Cong Wang, Ning Cao, Jin Li, Kui Ren, and Wenjing Lou. Secure Ranked Keyword Search over Encrypted Cloud Data. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 253–262. IEEE, 2010.

[133] Ming Li, Shucheng Yu, Ning Cao, and Wenjing Lou. Authorized Private Keyword Search over Encrypted Data in Cloud Computing. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 383–392. IEEE, 2011.

[134] Jianfeng Wang, Hua Ma, Qiang Tang, Jin Li, Hui Zhu, Siqi Ma, and Xiaofeng Chen. Efficient Verifiable Fuzzy Keyword Search over Encrypted Data in Cloud Computing. *Computer Science and Information Systems*, 10(2):667–684, 2013.

[135] Misspellings Detected by Google for the Query "Britney Spears". Online. Retrieved from http://www.google.com/jobs/archive/britney.html [Accessed 07/12/14].

[136] Shengyue Ji, Guoliang Li, Chen Li, and Jianhua Feng. Efficient Interactive Fuzzy Keyword Search. In *Proceedings of the 18th International Conference on World Wide Web*, pages 371–380. ACM, 2009.

[137] Chang Liu, Liehuang Zhu, Longyijia Li, and Yuan Tan. Fuzzy Keyword Search on Encrypted Cloud Storage Data with Small Index. In *Cloud Computing and Intelligence Systems (CCIS), 2011 IEEE International Conference on*, pages 269–273. IEEE, 2011.

[138] Ruixuan Li, Zhiyong Xu, Wanshang Kang, Kin Choong Yow, and Cheng-Zhong Xu. Efficient Multi-Keyword Ranked Query over Encrypted Data in Cloud Computing. *Future Generation Computer Systems*, 30:179–190, 2014.

[139] Bing Wang, Shucheng Yu, Wenjing Lou, and Y Thomas Hou. Privacy-Preserving Multi-Keyword Fuzzy Search over Encrypted Data in the Cloud. In *IEEE INFOCOM*, 2014.

[140] David Cash, Stanislaw Jarecki, Charanjit Jutla, Hugo Krawczyk, Marcel-Cătălin Roşu, and Michael Steiner. Highly-Scalable Searchable Symmetric Encryption with Support for Boolean Queries. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 353–373, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[141] Mark A. Will, Ryan K. L. Ko, and Silvino J. Schlickmann. Anonymous Data Sharing Between Organisations with Elliptic Curve Cryptography. In *Trustcom/BigDataSE/ICESS, 2017 IEEE*, pages 1024–1031. IEEE, 2017.

[142] Mark A. Will, Jeff Garae, Yu S. Tan, Craig Scoon, and Ryan K. L. Ko. Returning Control of Data to Users with a Personal Information Crunch - A Position Paper. In *2017 International Conference on Cloud Computing Research and Innovation (ICCCRI)*, pages 23–32, April 2017.

[143] Irit Dinur and Kobbi Nissim. Revealing Information While Preserving Privacy. In *Proceedings of the Twenty-second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '03, pages 202–210, New York, NY, USA, 2003. ACM.

[144] Pierangela Samarati and Latanya Sweeney. Protecting Privacy when Disclosing Information: K-Anonymity and its Enforcement through Gener-

alization and Suppression. Technical report, Technical report, SRI International, 1998.

[145] Latanya Sweeney. K-Anonymity: A Model for Protecting Privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(05):557–570, 2002.

[146] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. L-Diversity: Privacy Beyond K-anonymity. *ACM Trans. Knowl. Discov. Data*, 1(1), March 2007.

[147] Yarkın Doröz, Erdinç Öztürk, and Berk Sunar. Accelerating Fully Homomorphic Encryption in Hardware. *IEEE Transactions on Computers*, 64(6):1509–1521, 2015.

[148] Marc Tiehuis. libhcs. Online. Retrieved from https://github.com/tiehuis/libhcs [Accessed 10/08/17].

[149] Leo Ducas and Daniele Micciancio. FHEW: Fastest Homomorphic Encryption in the West. Online. Retrieved from https://github.com/lducas/FHEW [Accessed 21/03/17], 2014.

[150] Wilko Henecka, Ahmad-Reza Sadeghi, Thomas Schneider, Immo Wehrenberg, et al. TASTY: Tool for Automating Secure Two-Party Computations. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 451–462. ACM, 2010.

[151] Tool for Automating Efficient Secure Two-Party Computation Protocols. Online. Retrieved from https://github.com/encryptogroup/tasty [Accessed 30/06/18], 2016.

[152] Maarten H. Everts. JavaScript Proof-of-Concept Implementation of the Paillier Cryptosystem. Online. Retrieved from https://github.com/mhe/jspaillier [Accessed 08/08/17], 2013.

[153] W3C. Web Cryptography API. Online. Retrieved from https://www.w3.org/TR/WebCryptoAPI/ [Accessed 09/12/17], January 2017.

[154] Stanford E Taylor. Fluency in Silent Reading. *Taylor Associates/Communications Inc.*, 2004.

[155] Robert L Solso and Joseph F King. Frequency and Versatility of Letters in the English Language. *Behavior Research Methods & Instrumentation*, 8(3):283–286, 1976.

[156] Margaret Swain. *Needlework of Mary Queen of Scots*. Crowood, 2013.

[157] 109582 English Words. Online. Retrieved from http://www-01.sil.org/linguistics/wordlists/english/wordlist/wordsEn.txt [Accessed 04/12/14], 1991.

[158] Internet Activities Board. Ethics and the Internet. Online. Retrieved from https://www.ietf.org/rfc/rfc1087.txt [Accessed 04/12/14], January 1989.

[159] Python NGram 3.3 Documentation. Online. Retrieved from https://pythonhosted.org/ngram/ [Accessed 04/02/15].

[160] WinEdt Dictionaries. Online. Retrieved from http://www.winedt.org/Dict/ [Accessed 03/07/15].

[161] Lee-Feng Chien. Pat-Tree-Based Keyword Extraction for Chinese Information Retrieval. In *ACM SIGIR Forum*, volume 31, pages 50–58. ACM, 1997.

[162] Chih-Hao Tsai. Tsai's List of Chinese Words. Online. Retrieved from http://technology.chtsai.org/wordlist/tsaiword.zip [Accessed 01/02/15], 1996.

[163] Jovan Dj Golić. Cryptanalysis of Alleged A5 Stream Cipher. In *Advances in Cryptology—EUROCRYPT'97*, pages 239–255. Springer, 1997.

[164] Patrik Ekdahl and Thomas Johansson. A New Version of the Stream Cipher SNOW. In *Selected Areas in Cryptography*, pages 47–61. Springer, 2003.

[165] Martin Hell, Thomas Johansson, and Willi Meier. Grain: A Stream Cipher for Constrained Environments. *International Journal of Wireless and Mobile Computing*, 2(1):86–93, 2007.

[166] Matthew Robshaw and Olivier Billet. *New Stream Cipher Designs: The eSTREAM Finalists*, volume 4986. Springer, 2008.

[167] Mark A. Will and Ryan K. L. Ko. *Data Security in Cloud Computing*, chapter 5 – Distributing Encoded Data for Private Processing in the Cloud, pages 89–112. IET, 2017.

[168] Vignesh Ganapathy, Dilys Thomas, Tomas Feder, Hector Garcia-Molina, and Rajeev Motwani. Distributing Data for Secure Database Services. In *Proceedings of the 4th International Workshop on Privacy and Anonymity in the Information Society*, page 8. ACM, 2011.

[169] New Zealand Flag Referendums Bill. Online. Retrieved from http://www.legislation.govt.nz/bill/government/2015/0008/latest/096be8ed810db78b.pdf [Accessed 24/06/18], 2015.

[170] Tatsuaki Okamoto. Receipt-Free Electronic Voting Schemes for Large Scale Elections. In *International Workshop on Security Protocols*, pages 25–35. Springer, 1997.

[171] Feng Hao and Matthew Nicolas Kreeger. Every Vote Counts: Ensuring Integrity in Large-Scale DRE-based Electronic Voting. *IACR Cryptology ePrint Archive*, 2010:452, 2010.

[172] Paul C Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[173] INTERPOL. Overview. Online. Retrieved from https://www.interpol.int/About-INTERPOL/Overview [Accessed 16/02/17].

[174] Craig Scoon and Ryan KL Ko. The Data Privacy Matrix Project: Towards a Global Alignment of Data Privacy Laws. In *Trustcom/BigDataSE/ISPA, 2016 IEEE*, pages 1998–2005. IEEE, 2016.

[175] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

[176] John K Salmon, Mark A Moraes, Ron O Dror, and David E Shaw. Parallel Random Numbers: As Easy As 1, 2, 3. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.

[177] U Rajendra Acharya, P Subbanna Bhat, S Sitharama Iyengar, Ashok Rao, and Sumeet Dua. Classification of Heart Rate Data using Artificial Neural Network and Fuzzy Equivalence Relation. *Pattern Recognition*, 36(1):61–68, 2003.

[178] İnan Güler and Elif Derya Übeylı. ECG Beat Classifier Designed by Combined Neural Network Model. *Pattern Recognition*, 38(2):199–208, 2005.

[179] Josep Domingo-Ferrer and Vicenç Torra. A Critique of K-Anonymity and some of its Enhancements. In *Availability, Reliability and Security, 2008. ARES 08. Third International Conference on*, pages 990–993. IEEE, 2008.

[180] Mark A. Will and Ryan K. L. Ko. Computing Mod with a Variable Lookup Table. In Peter Mueller, Sabu M. Thampi, Md Zakirul Alam Bhuiyan, Ryan Ko, Robin Doss, and Jose M. Alcaraz Calero, editors, *Security in Computing and Communications*, pages 3–17, Singapore, 2016. Springer Singapore.

[181] Zhengjun Cao, Ruizhong Wei, and Xiaodong Lin. A Fast Modular Reduction Method. *IACR Cryptology ePrint Archive*, 2014:40, 2014.

[182] Chae Hoon Lim, Hyo Sun Hwang, and Pil Joong Lee. Fast Modular Reduction with Precomputation. In *Proceedings of Korea-Japan Joint Workshop on Information Security and Cryptology (JWISC'97)*, pages 65–79. Citeseer, 1997.

[183] Google. Privacy Policy. Online. Retrieved from http://static.googleusercontent.com/media/www.google.com/en//intl/en/policies/privacy/google_privacy_policy_en.pdf [Accessed 04/07/17], April 2017.

[184] Amazon Web Services. Data Privacy. Online. Retrieved from https://aws.amazon.com/compliance/data-privacy-faq/ [Accessed 04/07/17], 2017.

[185] Microsoft. Privacy & Cookies, Microsoft Online Services Privacy Statement. Online. Retrieved from https://www.microsoft.com/en-us/privacystatement/OnlineServices/Default.aspx [Accessed 04/07/17], August 2016.

[186] Google Drive Terms of Service. Online. Retrieved from https://www.google.com/drive/terms-of-service/ [Accessed 30/06/18], 2017.

[187] Dropbox Privacy Policy. Online. Retrieved from https://www.dropbox.com/privacy [Accessed 30/06/18], 2018.

Part III

Appendices

A

Custom Modulo Algorithm

The research presented in this appendix is a summary of the published paper [180], which tried to find an improvement or optimisation for a modulo algorithm. This was due to Hypothesis 1, as the modulus algorithm is heavily used in encryption schemes, any improvement would therefore make the encryption schemes themselves more practical. With the same modulo value being applied for encryption, precomputing values for the operation could help improve performance. The algorithm proposed in this appendix calculates the modulus, as shown in Algorithm 7 which computes $X \; mod \; Y$. The functions are shown before the pseudo code, where α denotes the parameter into the function. Lower case variables with a subscript represent bits at an index in the upper case variable, as shown by the definition of T. Finally, $\hat{}$ denotes the variable is an array or set. These notations will be the same for the theorems and proofs in Section A.2.

The bit shifting operation is the key feature of this algorithm, as it allows the use of a single precomputed value to find the modulus of all bits above the bit width of Y. We can also use multiple precomputed values in the form of a LUT and use a larger bit shift to improve performance, which is described in Section A.3. By using a LUT, actually allows the algorithm to be mostly implemented with FRIBs without much effort, as mentioned in Section 6.4.4.

A similar algorithm, proposed by Cao *et al.* [181] requires a precomputed value for each bit above the bit width of Y. This can be very costly as the precomputed values have a fixed size, where the algorithm proposed in this appendix can vary the number of precomputed values. The other key property

Algorithm 7 Custom mod algorithm

Compute: $X \bmod Y$

$Width(\alpha) = width\ of\ \alpha\ in\ terms\ of\ bits$
$Split(\alpha, w) = \{\sum_{i=0}^{w-1} \alpha_i 2^i, ..., \sum_{i=0}^{w-1} \alpha_{Width(\alpha)-w+i} 2^i\}$
$Num(\hat{\alpha}) = number\ of\ elements\ in\ \hat{\alpha}$
$T = \sum_{i=0}^{Width(T)-1} t_i 2^i,\ t_i \in \{0, 1\}$

1: $\hat{G} = Split(X, Width(Y))$
2: $N = Num(\hat{G}) - 1$
3: **while** $N > 0$ **do**
4: $\quad T = \hat{G}[N]$
5: $\quad$ **for** $i = Width(Y) - 1\ downto\ 0$ **do**
6: $\quad\quad T = T << 1$
7: $\quad\quad$ **while** $t_{Width(Y)} = 1$ **do**
8: $\quad\quad\quad t_{Width(Y)} = 0$
9: $\quad\quad\quad T = T + (2^{Width(Y)} \bmod Y)$
10: $\quad \hat{G}[N-1] = \hat{G}[N-1] + T$
11: $\quad$ **while** $\hat{G}[N-1]_{Width(Y)} = 1$ **do**
12: $\quad\quad \hat{G}[N-1]_{Width(Y)} = 0$
13: $\quad\quad \hat{G}[N-1] = \hat{G}[N-1] + (2^{Width(Y)} \bmod Y)$
14: $\quad N = N - 1$
15: **while** $\hat{G}[0] > Y$ **do**
16: $\quad \hat{G}[0] = \hat{G}[0] - Y$
return $\hat{G}[0]$

of our algorithm is that it only reads data of X once (reads an element in $\hat{G}$ once), starting from the uppermost bits and working down to the 0^{th} bit. This gives the algorithm excellent spatial and temporal locality. In terms of a hardware implementation, it also means the data can be streamed from memory. Then because only a fixed amount of data is read and computed at a time, allows custom hardware to be faster, have better area usage, and be more power efficient. This guaranteed data width is another property that other solutions cannot offer easily.

A.1 Description

Before the algorithm is described in further detail, first the functions must be explained.

- **Width:** Given the parameter α, this function will return the minimum number of bits in α. Put simply, it results the index of the uppermost high bit, plus one. For example, if 13 were inputted, the result would be 4.

- **Split:** The first parameter α is the value to be split, and the second w is a bit width value. This function splits α into a vector, so that each element is bit width w. If α cannot be be split up evenly (the bit width of α is not a multiple of w), then α can be padded with zero bits. For example, splitting 35 into a vector with an element bit width of 4, results in $\{3, 2\}$. In terms of binary values, such as $35 = 100011_2$, the result would be $\{0011_2, 0010_2\}$ (note the padded zeros at index 1).

- **Num:** Number of elements in the vector $\hat{\alpha}$ after the split function.

The first line of Algorithm 7 splits X into a vector $\hat{G}$ so that the elements have the same width as Y. Then we set N to the uppermost index of $\hat{G}$, because we will process the elements in reverse. Once we enter the loop, we will set T to element N for ease of understanding the algorithm.

The *For* loop will count from 0 to the number of bits in Y. Each iteration we shift T left by one, thus doubling the value T; if an overflow occurs, meaning that the width of T is no longer equal to that of Y (the bit at index $Width(Y)$ is high), we clear this overflow bit and add $2^{Width(Y)}\ mod\ Y$ to T. The value we add should be precomputed so that it is already in modulo Y; therefore, it costs just one addition operation. The reason for this is because if an overflow occurs, we are guaranteed that T is greater than Y; we add the modulo value of this overflow bit back to the answer, which keeps T the same bit width as Y. Therefore this means that T is or is close to the correct value. When adding $2^{Width(Y)}\ mod\ Y$ to T, another overflow may occur, which requires us to repeat line 7 until no overflow occurs. The LUT could be designed to handle two overflows meaning this repeating of line 7 is not required. Note

that clearing the overflow bit could also be achieved by subtracting Y from T. But depending on implementation, it is possible that multiple subtractions of Y are required to clear the overflow bit. Also subtraction would not allow the use of a lookup table.

Once we have finished the *For* loop, we have performed $Width(Y)$ number of shifts on T. We then add T to the next element in $\hat{G}$; it is possible for an overflow to occur at this step, so we have to include the loop on Line 11 to deal with this. Once we reach element 0–the lowermost bits of X–we know that we are already close to the correct result, meaning the loop can break. More subtractions could be required before the correct result is reached (depending on implementation).

The correct answer is now in element 0, which can be returned. In order to prove that this algorithm will produce the correct modulus answer, the theorems and proofs that the algorithm is based upon will now be discussed.

A.2 Theorems and Proofs

The algorithms main operation is to double the value of T, and by doing this, we are also doubling the modulus. This is shown in Theorem 1. Note that it is important to find the modulus of $2M$ because by doubling M, the result could become greater than Y.

Theorem 1 *If* $X \bmod Y = M$*, then* $2X \bmod Y = 2M \bmod Y$

Proof 1 *Given* $X \bmod Y = M$ *where* $X, Y \in \mathbb{Z}$ *and* $M \in \mathbb{Z}_0^Y$

$$\frac{X}{Y} = Q + M \quad (Q = Quotient)$$
$$\therefore X \bmod Y \equiv M \bmod Y$$

$$\frac{2X}{Y} = 2(Q + M)$$
$$\frac{2X}{Y} = 2Q + 2M$$
$$\therefore 2X \bmod Y \equiv 2M \bmod Y$$

Because an integer represented in binary is made up of powers of 2s, it is said that the number is the sum of power of 2s; therefore, if we take the modulus of each power of 2 and sum them, it is equivalent to summing the power of 2s then taking the modulus, as in the theorem below.

Theorem 2 *Given an integer X, which is the sum of power of 2s, then the sum of the modulus's of the power of 2s in Y, is equivalent to the modulus of X in Y.*

Proof 2 *Given $X \bmod Y$ where $X, Y \in \mathbb{Z}$*

$$\begin{aligned} X &= \sum_{i=0}^{n-1} x_i 2^i \quad \textit{(where } x_i \in \{0,1\}\textit{)} \\ \frac{x_i 2^i}{Y} &= Q_i + M_i \quad \textit{(where } Q_i, M_i \in \mathbb{Z}\textit{)} \\ x_i 2^i \bmod Y &\equiv M_i \bmod Y \\ \therefore X \bmod Y &\equiv \sum_{i=0}^{n-1} M_i \bmod Y \end{aligned}$$

Theorem 3 in the general context of the algorithm is the same as Theorem 1. However, Theorem 3 is required because the proposed algorithm can use a LUT which Theorem 1 does not make clear.

Theorem 3 *Given an integer X, which is to be shifted bit left by w, the modulus of X bit shifted by w is equivalent to finding the modulus of X after being bit shifted.*

Proof 3 *Given $(X << w) \bmod Y = M$ where $X, Y, M, w \in \mathbb{Z}$*

Because left bit shifting X by w is equivalent to doubling X w times, we can use Theorem 1 to prove that:

$$((X \bmod Y) << w) \bmod Y \equiv (X << w) \bmod Y \equiv M$$

The first three theorems are the basic underlying operations used in the algorithm. Now the proof for the fundamental idea behind the algorithm is

required. Theorem 4 proves the initial step of the algorithm, splitting up X into elements of an array/vector so that they can be concatenated together to give X. Then when finding the modulus of X, the modulus of each element (with bit shifting) is summed.

Theorem 4 *When finding $X \bmod Y$, if X is larger then Y in terms of number of bits, the modulus is equivalent to dividing X into elements of the same bit size as Y, follow by the bit shifting and summing the modulus values of each element.*

Proof 4 *Given $X \bmod Y = M$ where $X, Y, M \in \mathbb{Z}$*

Because the concatenation of $\hat{X}$ is equivalent to X. Then by shifting each item in $\hat{X}$ to the correct bit position and summing, is equivalent to X; therefore, we can use Theorem 2 and 3 so that

$$\sum_{i=0}^{(n/k)-1} (\hat{X}_i << ki) \bmod Y \equiv X \bmod Y$$

Note: if $n \bmod k \neq 0$, then X can be padded with $0s$ until $n \bmod k = 0$.

Theorem 5 proves that a single precomputed value (or a LUT), can be used to help find the modulus in each element greater than 0. This is an important property of the algorithm, as it allows the use of more precomputed values in the form of a lookup table.

Theorem 5 *When calculating the modulus off $\hat{X}_i$ in Y where $i > 0$, the modulus is equivalent to multiplying $\hat{X}_i$ by $\left(2^k \bmod Y << k(i-1)\right)$.*

Proof 5

$$\begin{aligned} & \hat{X}_i << ki \bmod Y \\ \equiv\ & \hat{X}_i 2^{ki} \bmod Y \\ \equiv\ & (\hat{X}_i \bmod Y) \times (2^{ki} \bmod Y) \\ \equiv\ & (\hat{X}_i \bmod Y) \times (2^k << k(i-1) \bmod Y) \end{aligned}$$

Instead of multiplying (as in Theorem 5), we instead shift each individual bit in an element and add if an overflow occurs. These are proven to be equivalent in Theorem 6.

Theorem 6 *Left shifting $\hat{X}_i$ by ki in modulo Y where $i > 0$, is equivalent to shifting $\hat{X}_i$ one shift at a time. Then, if the kth bit becomes high after any shift (or add operation), drop it, and add $2^k \bmod Y$ to $\hat{X}_i$. Therefore, keeping the bit width of $\hat{X}_i$ constant, while remaining in modulo Y.*

Proof 6

$$\begin{aligned} & \hat{X}_i \times 2^k i \bmod Y \\ \equiv\ & \hat{X}_i << ki \bmod Y \\ \equiv\ & (((\hat{X}_i << 1 \bmod Y) \ldots) << 1 \bmod Y) \end{aligned}$$

After each shift we put $\hat{X}$ back into modulo Y, and because $\hat{X}$ has k bits, if the kth bit becomes high (an overflow), we know that $\hat{X}$ is definitely bigger than Y. Then by using Theorem 4 and 5, we can say that $\hat{X}$ is equivalent in modulo Y to adding $2^k \bmod Y$ to $\hat{X}$ (after dropping the kth bit).

Theorem 6 is the main theorem for this algorithms operation (producing the correct answer); however, for implementations would suffer from performance issues. This is due to the amount of shifting and potential adding which we would need to perform. So instead of shifting an element to its correct position directly, we can just shift it by the number of bits in Y, then start shifting the next element as well. This is shown in Theorem 7, and in terms of performance of the algorithm, is the most important theorem.

Theorem 7 *If we start calculating the modulus of X in Y at $\hat{X}_{(n/k)-1}$, then instead of performing $k\,(((n/k)-1)-1)$ number of shifts, we can perform k shifts, then add $\hat{X}_{(n/k)-1}$ to $\hat{X}_{(n/k)-2}$. Repeating until we reach $\hat{X}_0$ which will contain the result.*

Proof 7

$$\begin{aligned} & \left(\hat{X}_{(n/k)-1} << k((n/k)-2)\right) + \left(\hat{X}_{(n/k)-2} << k((n/k)-3)\right) \\ \equiv\ & \left(\hat{X}_{(n/k)-1} << k << k((n/k)-3)\right) + \left(\hat{X}_{(n/k)-2} << k((n/k)-3)\right) \\ \equiv\ & \left(\left(\hat{X}_{(n/k)-1} << k\right) + \left(\hat{X}_{(n/k)-2}\right)\right) << k((n/k)-3) \end{aligned}$$

A.2.1 Example

Below is an example on how to solve 1620 *mod* 11 on a 4-bit processor using the proposed algorithm.

$$
\begin{aligned}
1620 &= 011001010100_2 \\
11 &= 1011_2 \\
\hat{G} &= \{0110_2,\ 0101_2,\ 0100_2\}
\end{aligned}
$$

$$
\begin{aligned}
T &= \hat{G}[2] \\
&= 0110_2 \\
T &= T << 1 \\
&= 1100_2 \\
T &= T << 1 \\
&= 1\ 1000_2 \\
T &= 1000_2 + \\
&\quad (10000_2\ mod\ 1011_2) \\
&= 1000_2 + 0101_2 \\
&= 1101_2 \\
T &= T << 1 \\
&= 1\ 1010_2 \\
T &= 1010_2 + 0101_2 \\
&= 1111_2 \\
T &= T << 1 \\
&= 1\ 1110_2 \\
T &= 1110_2 + 0101_2 \\
&= 1\ 0011_2 \\
&= 0011_2 + 0101_2 \\
&= 1000_2 \\
T &= T + \hat{G}[1] \\
&= 1000_2 + 0101_2 \\
&= 1101_2
\end{aligned}
$$

$$
\begin{aligned}
T &= T << 1 \\
&= 1\ 1010_2 \\
T &= 1010_2 + 0101_2 \\
&= 1111_2 \\
T &= T << 1 \\
&= 1\ 1110_2 \\
T &= 1110_2 + 0101_2 \\
&= 1\ 0011_2 \\
T &= 0011_2 + 0101_2 \\
&= 1000_2 \\
T &= T << 1 \\
&= 1\ 0000_2 \\
T &= 0000_2 + 0101_2 \\
&= 0101_2 \\
T &= T << 1 \\
&= 1010_2 \\
T &= T + \hat{G}[0] \\
&= 1010_2 + 0100_2 \\
&= 1110_2 \\
T &= T - 11 \\
&= 1110_2 - 1011_2 \\
&= 0011_2
\end{aligned}
$$

$$\therefore \mathbf{1620\ mod\ 11 = 3}$$

Table A.1: Simple modulo LUT for a single bit

Key	Value
0	0
1	$2^k \ mod \ Y$

Table A.2: Simple modulo LUT for two bits

Key	Value
00	0
01	$2^k \ mod \ Y$
10	$2^{k+1} \ mod \ Y$
11	$(2^k + 2^{k+1}) \ mod \ Y$

A.3 Lookup Table

A useful property of this algorithm is the use of a variable sized LUT. LUTs may not be suitable for all applications or functions, such as key generation, because the overhead in creating the table could be too expensive. However, for applications where large amounts of data must be computed with the same modulo value, such as smart-cards or secure tunnels, there is the potential of a performance gain. The algorithm in its most basic form already uses a LUT. For each shift, if an overflow occurs a precomputed value is added, else add 0, as shown in Table A.1.

This can be extended to look at more bits at once. For example, looking at two bits, the LUT would be that of Table A.2. By shifting two bits at a time (instead of a single shift), still only requires a single add. However, this makes a software implementation slightly more difficult, because the overflow bit cannot be used anymore; therefore, the upper two bits of T need to be looked at before shifting. The proof-of-concept software implementation used an *AND* operation then a shift to get these upper bits. On a 64-bit processor, a LUT with a key size of up to 64-bits, only requires the *AND* and shift operations to be executed on a single 64-bit register, regardless of the size of Y. To possibly improve performance even more, one approach that could be explored is reversing the bits in each element (and keys), meaning only an *AND* operation would be required.

Given that the LUT is of variable size, it is important to make the whole table fit into the processors cache. This makes the lookup time require significantly less clock cycles than fetching from main memory, thus improving performance. The size of Y has the biggest impact on the size of the lookup table. For example, if Y is a 2048-bit value then each item in the table requires

2048-bits, plus the number of bits for each key.

A.4 Concluding Remarks

As mentioned, an algorithm similar to the one proposed in this appendix is the Fast Modular Reduction Method proposed by Cao *et al.* [181]. Their method also uses a fixed size LUT similar to precomputed tables described in [182]. Comparing these tables, the Fast Modular Reduction Method [181] has a fixed size lookup table. So for example, if Y is 2048 bits, then there must be 2048 entries in the table, each of a size of 2048 bits. Resulting in a size of approximately 4Mb. Also because of this, the bit width of the input X, can be no more than double the bit width of Y. This is because the lookup table only contains entries for the bits up to double that of Y, which for this example is 2^{2048} to 2^{4096}. This is a major limitation of their algorithm. However, when looking at the LUT proposed in this appendix, it can vary in size and can support an arbitrary bit length of X. This is important to allow the table to fit in cache, and for devices which have limited storage. Using the same example where Y is 2048 bits, if the LUT key is 8-bits, then the total size is approximately 0.5Mb.

An implementation in hardware, for example an FPGA would yield faster results than software; however, performance in software did not improve over other modular algorithms. Even though these algorithms have already been highly optimised, any improvement in the implementation of the proposed algorithm still would not have made fully homomorphic encryption practical. Investigating the modular operation in detail highlighted the challenges of making fully homomorphic encryption practical with current schemes, thus leading from Hypothesis 1 to Hypothesis 2.

B

A Simple Fully Homomorphic Encryption Algorithm

Limited improvements were realised by looking into already highly optimised large number operations, as detailed in Appendix A when trying to answer Hypothesis 1; therefore, the fully homomorphic encryption algorithms themselves need to be made more lightweight in order to achieve practical performance (Hypothesis 2). We started with a simple equation using waves, in an attempt to see what is required to make it work and to make it at least secure against brute-force attacks. This work should not be considered unique but an exploratory process into Hypothesis 2.

B.1 General Concept

To support addition and multiplication, continuous waves were used to encrypt data; such as the cosine wave. Given $m = a\cos(fx) + b$, a is the amplitude, f is the frequency, b is the displacement of the wave, while m (y) is the message (in reality $m = a\cos(fx) + b \equiv m = ax + b$). For addition, we can keep a, f and x private, and b as the cipher text. An example is shown below where $x = 3$, $f = 2$ and $a = 10$.

$$E(5) \rightarrow 5 = 10\cos(2 \times 3) + b \qquad E(6) \rightarrow 6 = 10\cos(2 \times 3) + b$$
$$b = 5 - 10\cos(6) \qquad b = 6 - 10\cos(6)$$
$$= -4.6017 \qquad = -3.6017$$

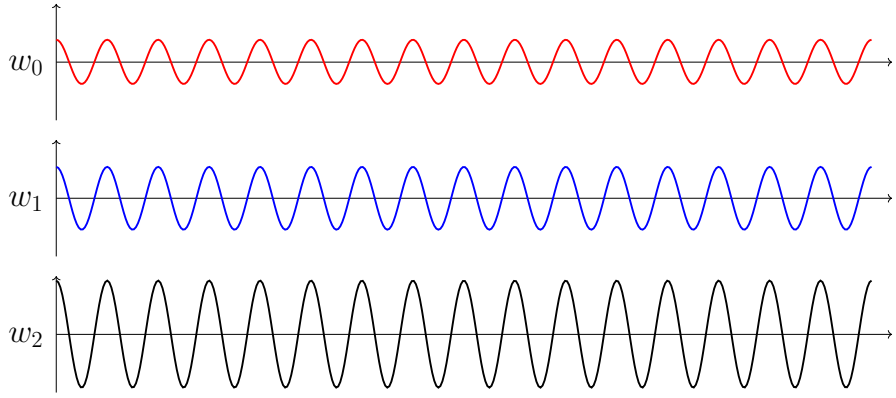

Figure B.1: Visual representation of adding two waves together ($w_0 + w_1 = w_2$)

A problem can be seen from this example, where the difference between the two m's and b's are both 1. To add randomness, the value a can be selected at random; however, it needs to be combined with b to form the cipher-text value.

$$E(8) \rightarrow 8 = 19\cos(4 \times 5) + b \qquad E(11) \rightarrow 11 = 27\cos(4 \times 5) + b$$
$$b = 0.24644 \qquad b = -0.018216$$
$$\therefore E(8) = 19, 0.24644 \qquad \therefore E(11) = 27, -0.018216$$

Adding the values together gives 46, and 0.228224, which can be decrypted to $19 \rightarrow y = 46cos(4 \times 5) + 0.228224$; where a visual representation is given in Figure B.1. The waves themselves would only be important if x was random; however, these would need to be tracked or if the data was stored in a physical form such as light waves being added together during transit.[1]

Multiplication is more difficult because a new value is introduction; for example, $(a_0x + b_0)(a_1x + b_1)$ now contains x^2. In the example using cos, the growth needs to be controlled such that $\cos^2$ is represented as cos; $\cos^2(4 \times 5) \equiv \cos(20) - 0.24155109$. A more precisely chosen frequency would yield $\cos^2(fx) \equiv cos(fx) + i$ where $i \in \mathbb{Z}$. Therefore, during multiplication instead of adding $\cos^2$, the values $\cos(20) - 0.24155109$ would be added, as shown below. An issue that starts to arise is precision, but this can be left as an open problem for now as using a different equation (not waves) would help solve this.

[1]The idea of computation in a physical form was the original reason for using waves.

$$(27\cos(fx) - 0.018216)(19\cos(fx) + 0.24644)$$
$$513\cos^2(fx) - 0.346104\cos(fx) + 6.65388\cos(fx) - 0.0044891504$$
$$513\cos^2(fx) + 6.307776\cos(fx) - 0.0044891504$$
$$513(\cos(fx) - 0.24155109) + 6.307776\cos(fx) - 0.0044891504$$
$$519.307776\cos(fx) - 123.92019817$$

The value -0.24155109 now needs to be protected, which can be done in the same manner where $-0.24155109 = a\cos(20) + b$; setting $a = 15$ results in $b = -6.362782017$.

$$513(15\cos(fx) - 6.362782017) + 6.307776\cos(fx) - 0.0044891504$$
$$8214.307776\cos(fx) - 3264.111664$$

The value a can also be protected, using a different fx. In fact, all this achieves is changing the unknown part from $\cos(fx)$ to $\cos(fx_0)/\cos(fx_1)$. For example, if $fx_0 = 5$ and $fx_1 = 2$, then below is an example encryption for 10 and 12. Repeating this step would then be $\cos(fx_0)/\cos(fx_1)/\cos(fx_2)$ and can be done indefinitely. It does not change the fact that to break the values, only one part (λ) needs to be guessed ($m = a(\lambda) + b$ where a and b are known), but it does increase the number of possibilities. The value used for multiplication to stop new values being added now becomes $\cos^2(fx_0)/\cos^2(fx_1)$, where in the example below $\cos^2(5)/\cos^2(2) = 0.4646$. This value is protected again where $0.4646 - 14\cos(5) = -3.5066$ and $14\cos(2) = -5.826$ (note that 14 was chosen at random).

$$E(10) \rightarrow 10 - 25\cos(5) = 2.908445 \qquad E(12) \rightarrow 12 - 31\cos(5) = 3.206$$
$$25\cos(2) = -10.40367 \qquad 31\cos(2) = -12.90055$$
$$\therefore E(10) = -10.40367, 2.908445 \qquad \therefore E(12) = -12.90055, 3.206$$

$$
\begin{aligned}
E(10) + E(12) &= -23.304, 6.114445 \\
D() &= -23.304/\cos(2) \times \cos(5) + 6.114445 \\
&= 22
\end{aligned}
$$

$$
\begin{aligned}
E(10) \times E(12) &= (-10.40367x + 2.908445)(-12.90055x + 3.206) \\
&= 134.2130650185x^2 - 70.8747061648x + 9.32447467 \\
&= 134.2130650185(-5.826x - 3.5066) - 70.8747061648x \\
&\quad + 9.32447467 \\
&= -781.9253167978x - 470.6315337939 - 70.8747061648x \\
&\quad + 9.32447467 \\
&= -852.8000229626x - 461.3070591239 \\
D(E(10) \times E(12)) &= -852.8000229626(cos(5)/cos(2)) - 461.3070591239 \\
&= 120
\end{aligned}
$$

B.2 Outcome

If two equal values are combined and solved using simultaneous equations, then the plain-text value is revealed; therefore, it can be easy to brute force no matter the complexity of the equation or how big the numbers are. This is the main issue with many simple examples of fully homomorphic encryption, and like these many examples of fully homomorphic encryption, error can be added into the equations. However, this error needs to be managed; for the current method of waves, where precision errors are already problematic, it would be difficult to control the error. Like lattice based schemes, the point could be shifted off the wave, where the closest point to the wave is the value. Ultimately, using waves did not give an advantage over a simple linear equation, because supporting both addition and multiplication limited their usefulness; however, this work did help lead onto the main research presented in this thesis, in particular Chapter 4.

Even though this would be a symmetric scheme, for cloud processing this would be acceptable; however, the other limitations were too great to continue along this path. The primary outcome from this work was the ability to add search for encrypted characters, which have not had any homomorphic processing over them. To accomplish this, characters can be encrypted such that a predefined selection of inputs have a range of a values. For example, encrypting a set of characters m,n,o with a between $1,000 - 10,000$, would mean the cipher values are similar to each other. The set p,q,r could have a range a value between $100,000 - 110,000$, which would be differentiate them from m,n,o. This idea of searching based on cipher ranges was explored in Chapter 4.

C

Screenshots

This appendix contains larger versions of screenshots for the following figures:

Chapter 3: Extending to Survey System

- Figure 3.3 (page 49) to Figure C.1 (page 228): Generating the public and private key for a new survey
- Figure 3.4 (page 49) to Figure C.2 (page 229): Interface for creating a survey
- Figure 3.5 (page 51) to Figure C.3 (page 230): Example of a user taking a survey, with the encryption occurring in the background
- Figure 3.6 (page 52) to Figure C.4 (page 231): Example of a user reviewing the results of their survey
- Figure 3.7 (page 52) to Figure C.5 (page 232): Example of how the cipher and plain-text results for a survey are presented to the user
- Figure 3.8 (page 53) to Figure C.6 (page 233): Example of the performance for decrypting each answer of a survey

Chapter 7: Proof-of-Concept Search Implementation

- Figure 7.7 (page 165) to Figures C.7 and C.8 (pages 234 and 235): A proof-of-concept screenshot of searching for tagged wallet ID

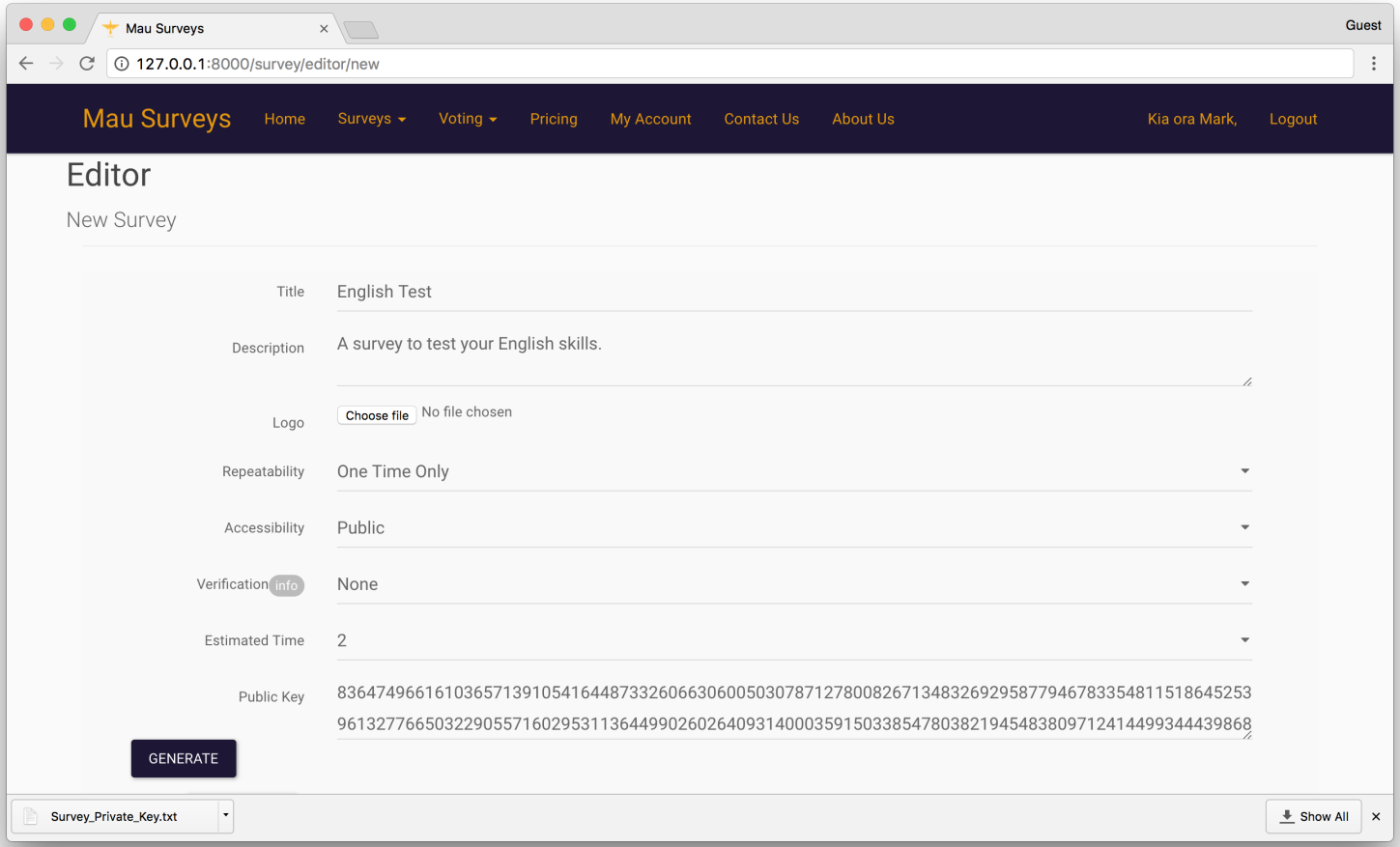

Figure C.1: Generating the public and private key for a new survey

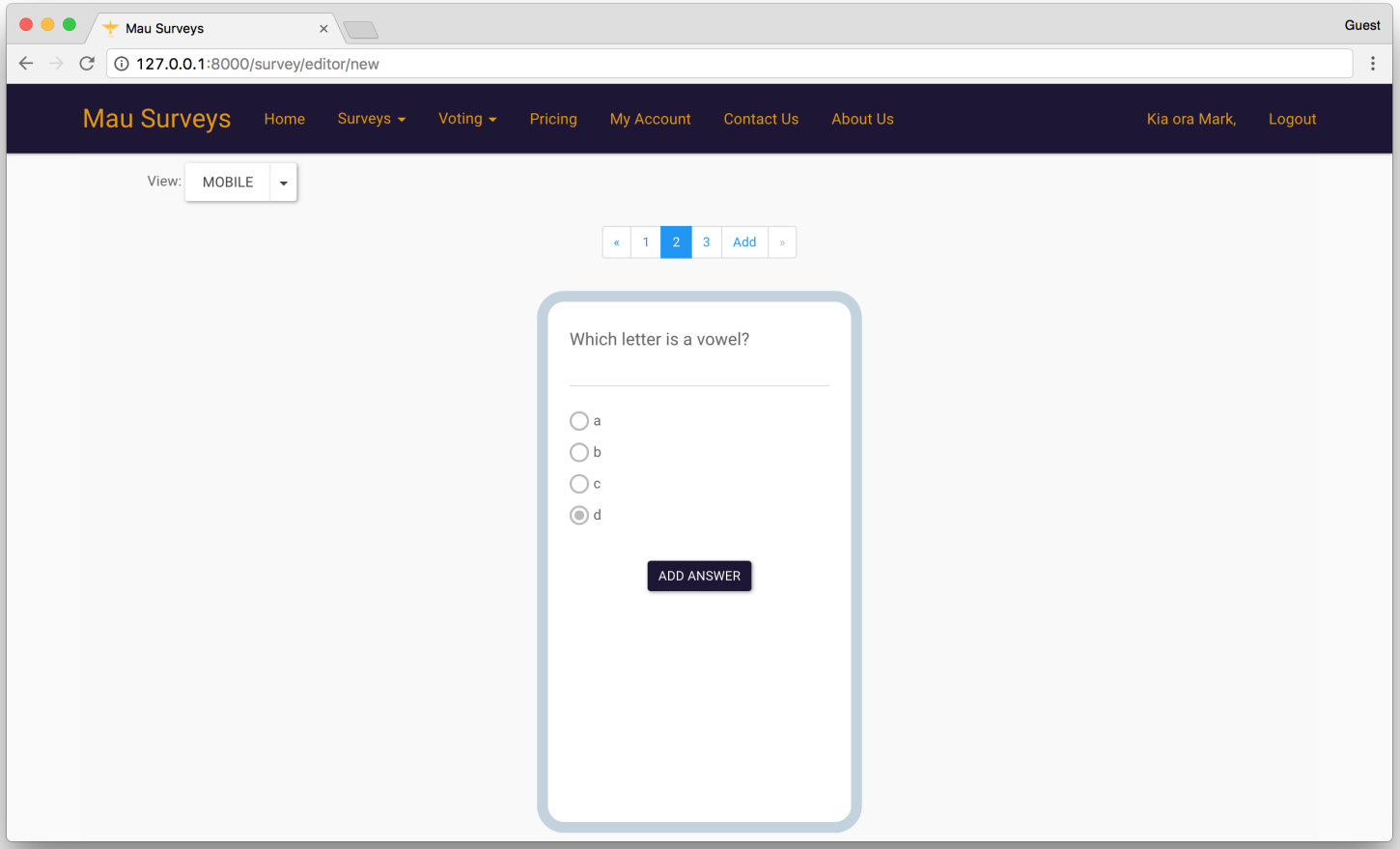

Figure C.2: Interface for creating a survey

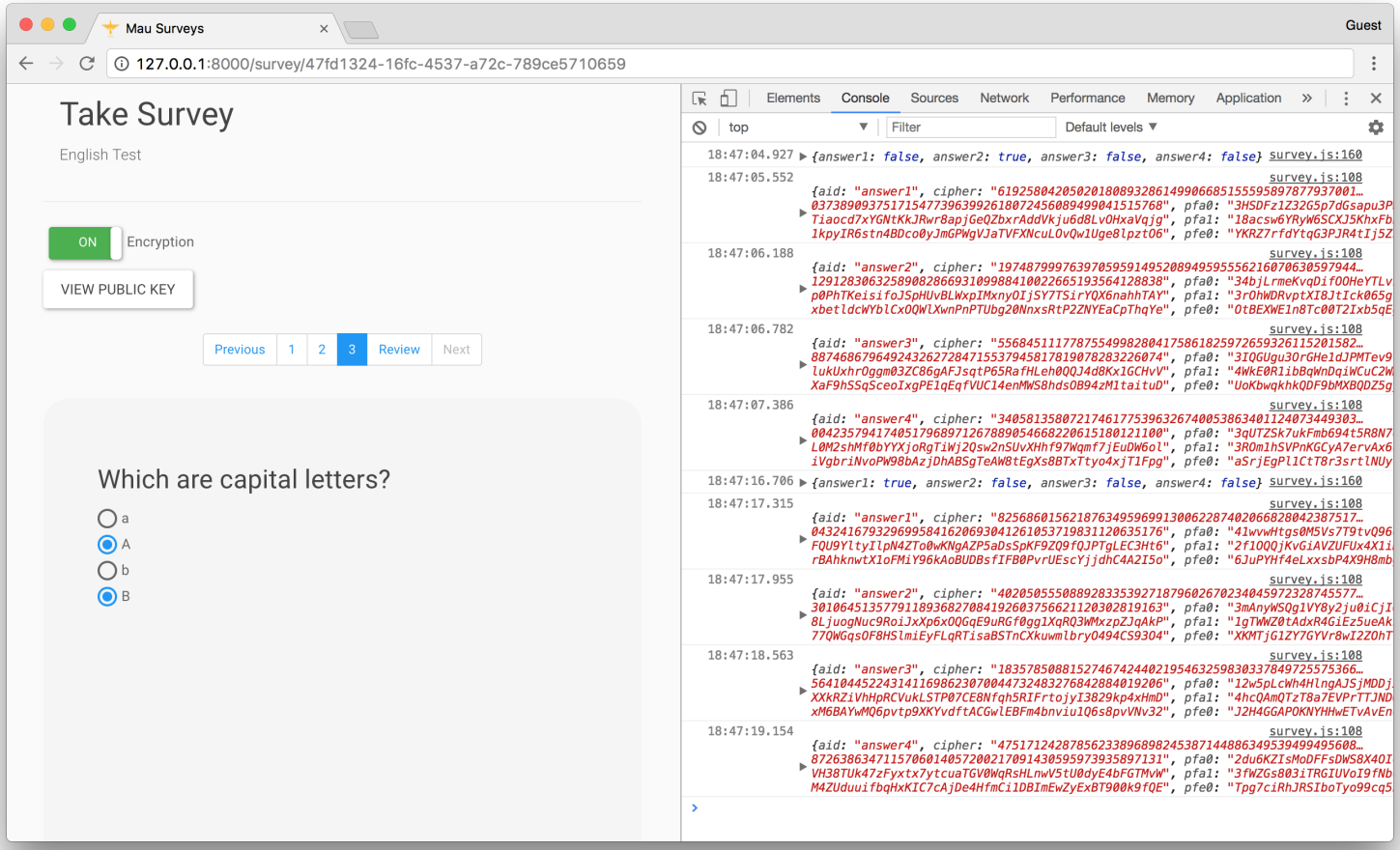

Figure C.3: Example of a user taking a survey, with the encryption occurring in the background

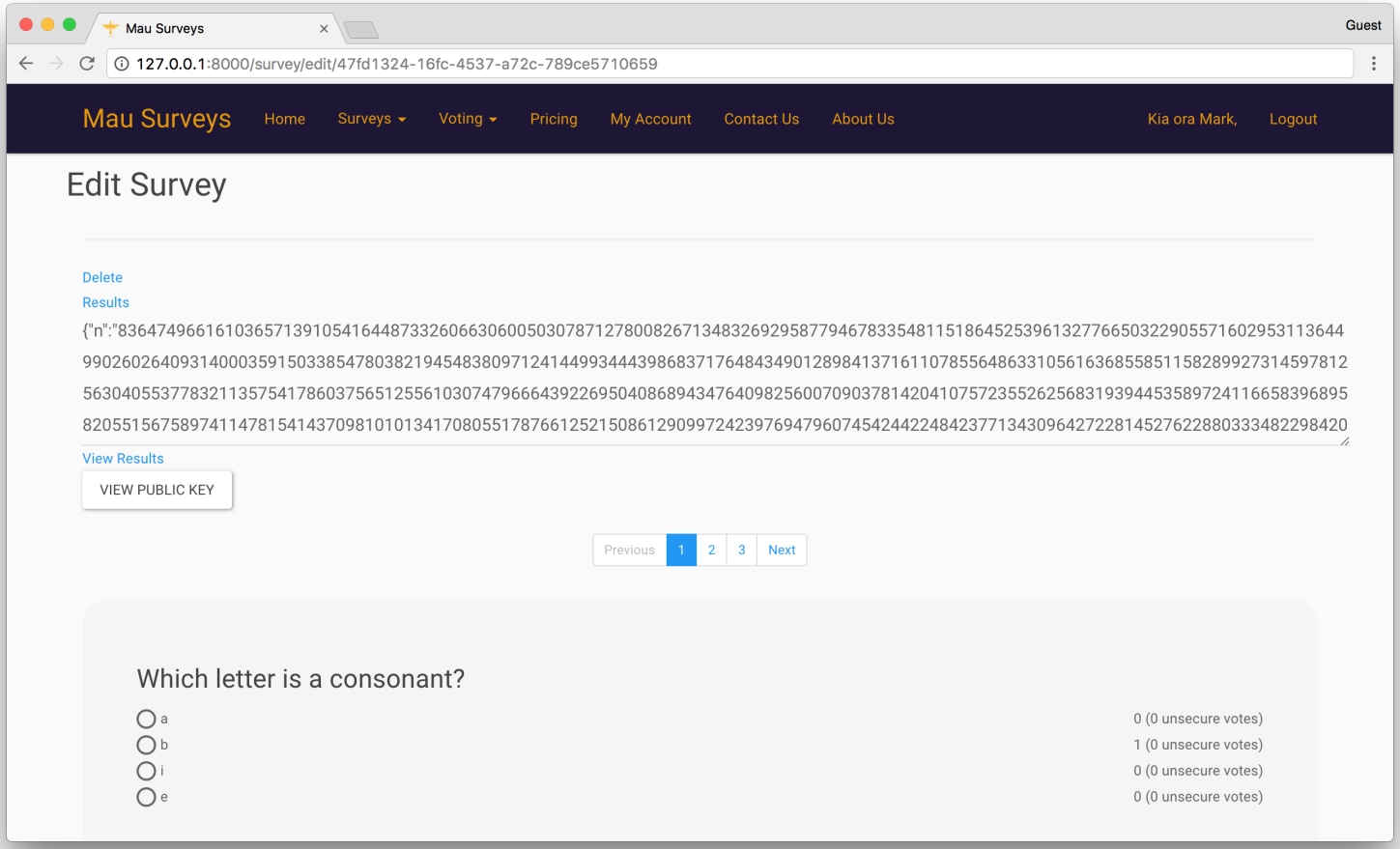

Figure C.4: Example of a user reviewing the results of their survey

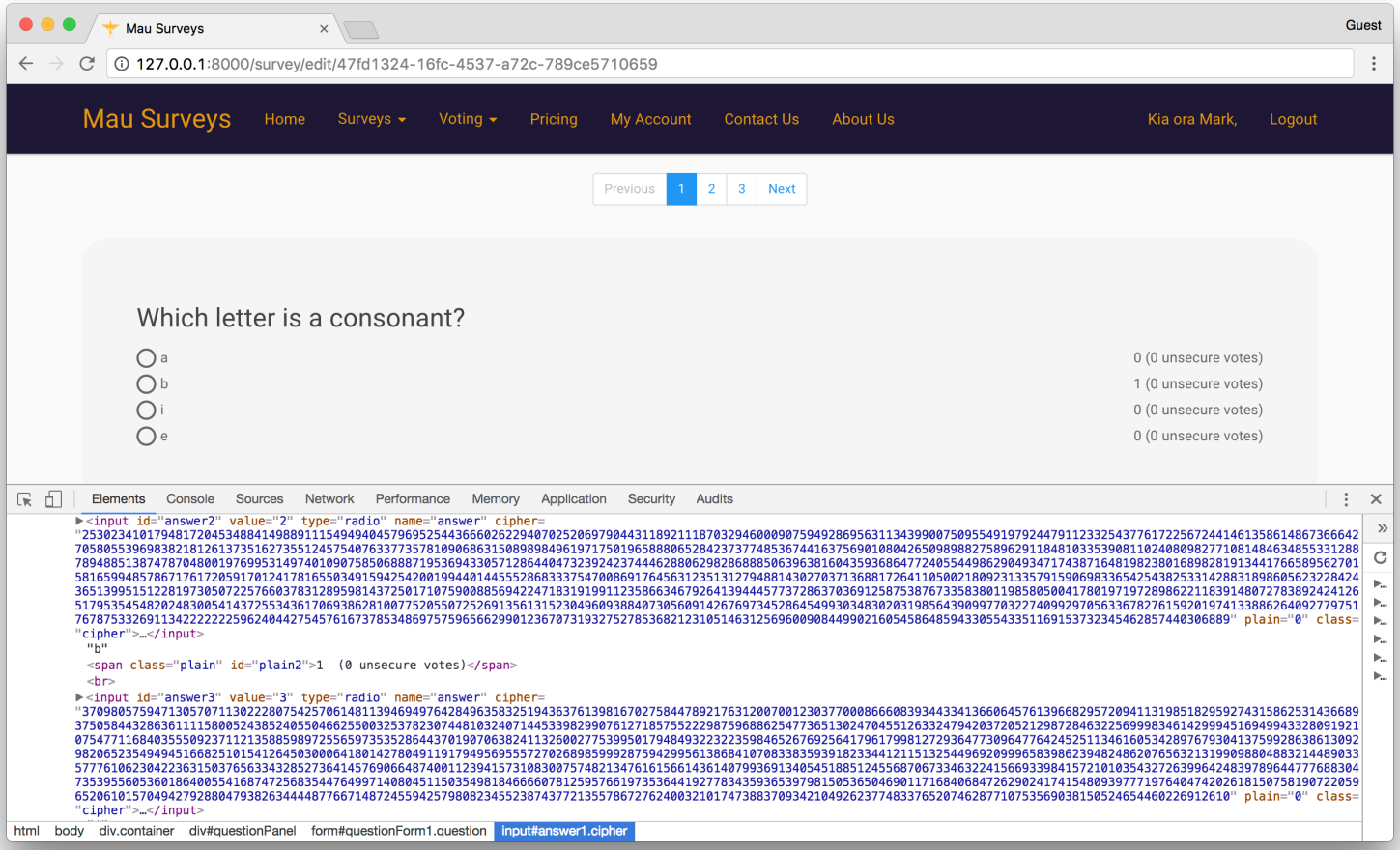

Figure C.5: Example of how the cipher and plain-text results for a survey are presented to the user

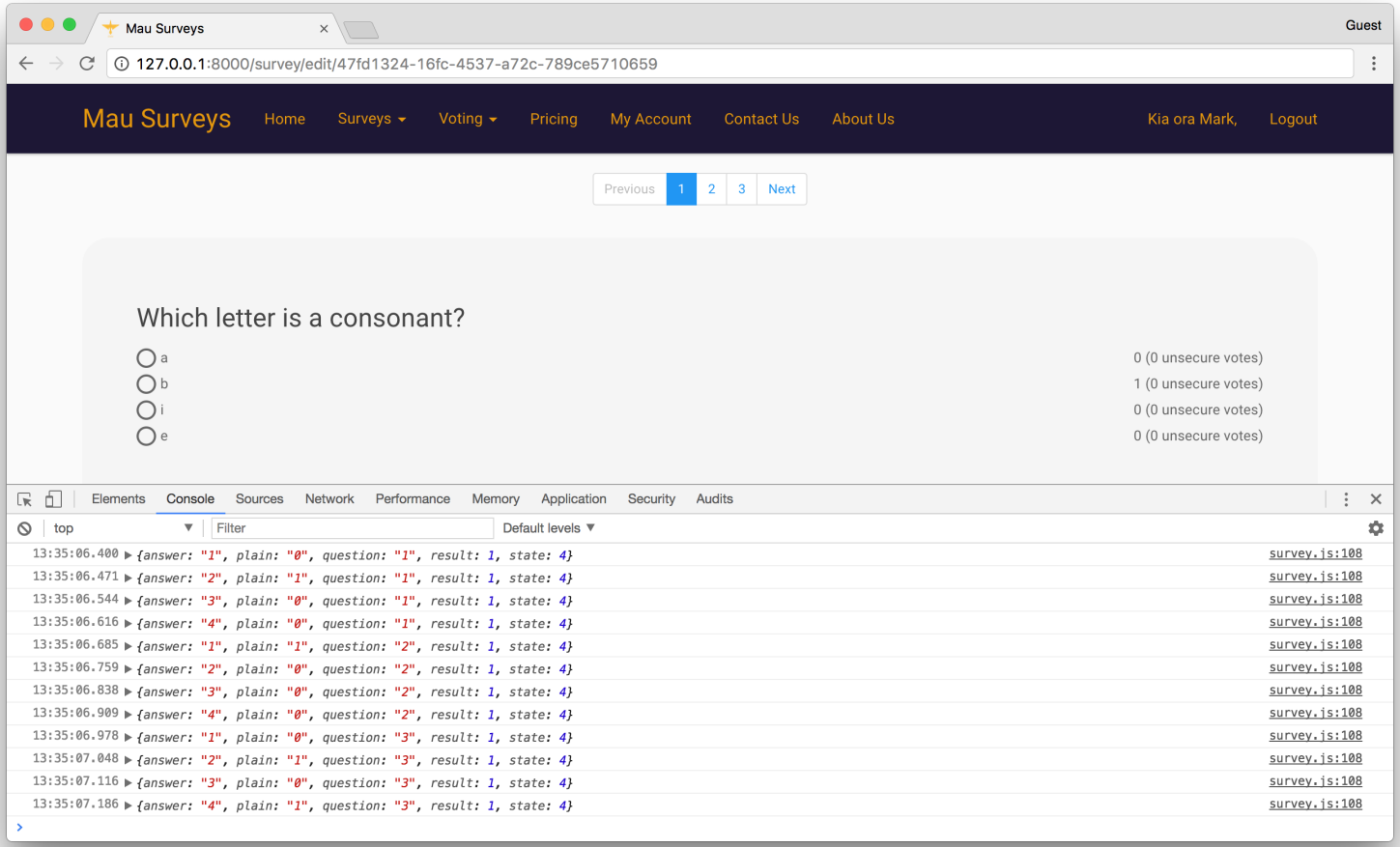

Figure C.6: Example of the performance for decrypting each answer of a survey

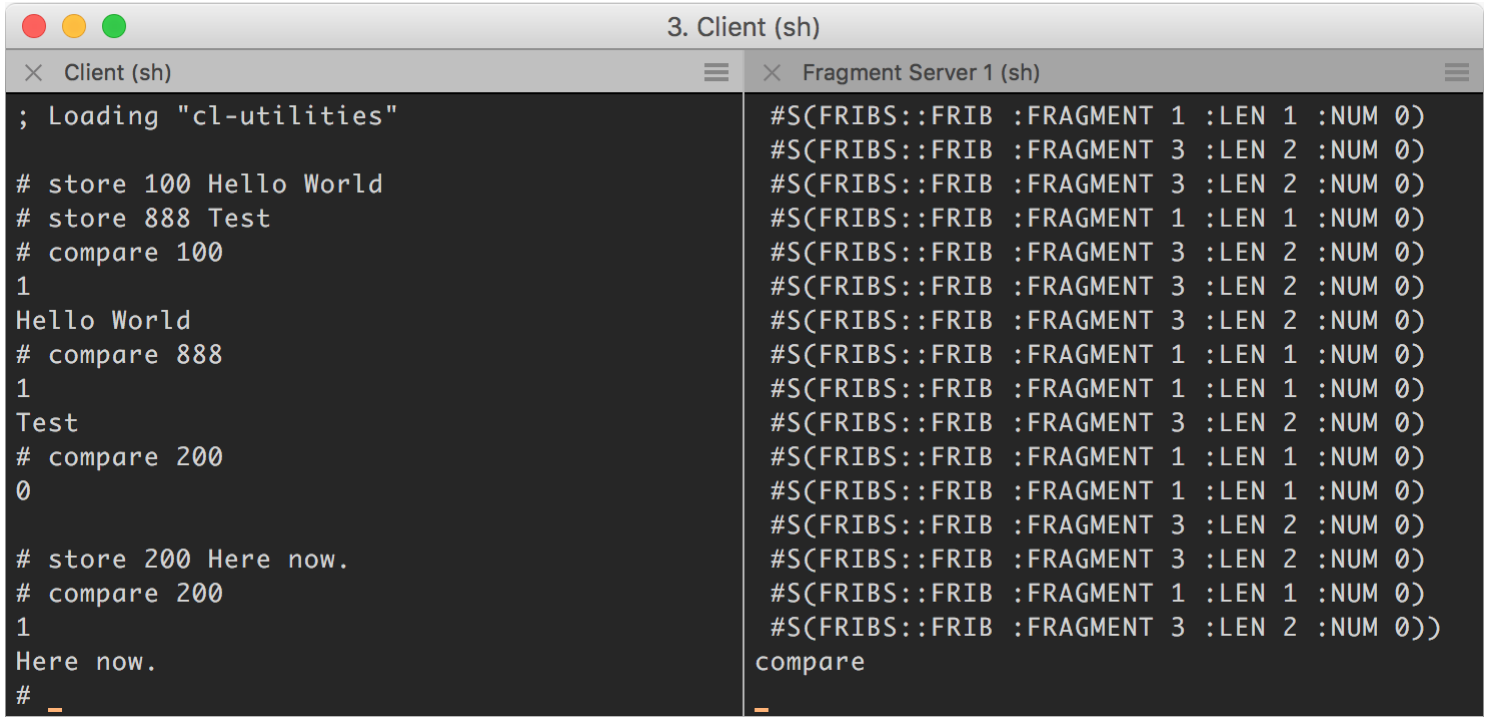

Figure C.7: A proof-of-concept screenshot of searching for tagged wallet ID (top two sessions)

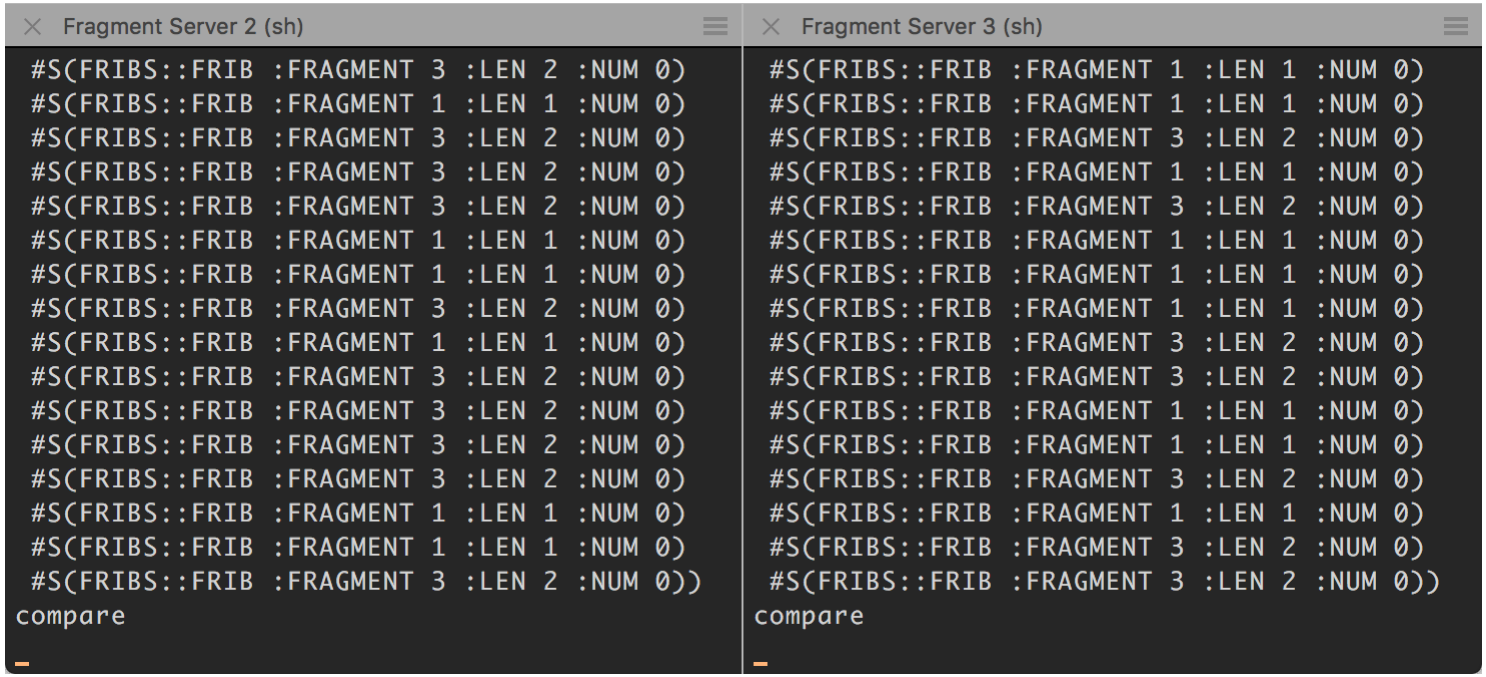

Figure C.8: A proof-of-concept screenshot of searching for tagged wallet ID (bottom two sessions)

D

Source Code

The source code for proof-of-concept implementations given in this thesis can be found at the following locations:

- https://github.com/maw41/FRIBs_Voting
- https://github.com/maw41/FRIBs_Search

Access to the following repository can be requested, for the web implementation of Mau Surveys.

- https://github.com/maw41/SecureSurveys

E

Data Privacy Law Examples

This appendix provides a few examples of country laws and cloud service polices in regards to data privacy.

E.1 Country Law

Part 2 of the New Zealand Privacy Act 1993, Principle 5 details storage and security of personal information for an agency to meet and is given below. Ultimately this part is vague in the sense that *reasonable* could range from out-of-the-box security, to access policies and full data encryption.

> An agency that holds personal information shall ensure—
>
> (a) that the information is protected, by such security safeguards as it is reasonable in the circumstances to take, against—
>
> (i) loss; and
>
> (ii) access, use, modification, or disclosure, except with the authority of the agency that holds the information; and
>
> (iii) other misuse; and
>
> (b) that if it is necessary for the information to be given to a person in connection with the provision of a service to the agency, everything reasonably within the power of the agency is done to prevent unauthorised use or unauthorised disclosure of the information.

In terms of lawfully requesting data, in Principle 11 Part 2, (e) on non-compliance disclosure is important as it shows that law enforcement agencies can request or get court orders for information. Section (f) mentions public health and safety, which includes risks such as terrorism, but could also be misused. The same is described in the United Kingdom Data Protection Act 1998; Sections 28 and 29 allow an organisation (cloud service) to disclose data if they are persuaded in the interests of national security or criminal investigations; therefore, they can choose to withhold information, unless ordered by the court. This is problematic, as the vagueness of the law does lead to concerns and potential loopholes. For majority of cloud users, both personal and enterprise, the probability of their data being requested is low. With FRIBs distributing data–potentially across different jurisdictions–then separate judges and other law personal need to approve the data request, thus distributing data privacy in terms of law as well.

E.2 Cloud Service Terms and Conditions

The three main cloud service providers include in their polices to abide by the law for data requests; however, it depends in which jurisdiction the data resides.

Google Drive Privacy Policy [183]

> We will share personal information with companies, organizations or individuals outside of Google if we have a good-faith belief that access, use, preservation or disclosure of the information is reasonably necessary to:
>
> - meet any applicable law, regulation, legal process or enforceable governmental request.
> - enforce applicable Terms of Service, including investigation of potential violations.
> - detect, prevent, or otherwise address fraud, security or technical issues.
> - protect against harm to the rights, property or safety of Google, our users or the public as required or permitted by law.

Amazon AWS Data Privacy [184]

> We are vigilant about our customers' privacy. We do not disclose customer content unless we're required to do so to comply with the law or a valid and binding order of a governmental or regulatory body. Governmental and regulatory bodies need to follow the applicable legal process to obtain valid and binding orders, and we review all orders and object to overbroad or otherwise inappropriate ones. Unless prohibited from doing so or there is clear indication of illegal conduct in connection with the use of Amazon products or services, Amazon notifies customers before disclosing customer content so they can seek protection from disclosure. It's also important to point out that our customers can encrypt their customer content, and we provide customers with the option to manage their own encryption keys.

Microsoft Online Services [185]

> Microsoft will not disclose Customer Data to law enforcement unless required by law. Should law enforcement contact Microsoft with a demand for Customer Data, Microsoft will attempt to redirect the law enforcement agency to request that data directly from you. If compelled to disclose Customer Data to law enforcement, then Microsoft will promptly notify you and provide you a copy of the demand unless legally prohibited from doing so.

The above policies are for data requests from law enforcement; however, the same companies also state they have rights to user data, which can be seen as a breech of data privacy. FRIBs would prevent a cloud service from accessing and using data uploaded to their service.

Google Drive [186]

> When you upload, submit, store, send or receive content to or through Google Drive, you give Google a worldwide license to use, host, store, reproduce, modify, create derivative works (such as those resulting from translations, adaptations or other changes we make so that your content works better with our services), communicate,

publish, publicly perform, publicly display and distribute such content. The rights you grant in this license are for the limited purpose of operating, promoting, and improving our services, and to develop new ones. This license continues even if you stop using our services unless you delete your content. Make sure you have the necessary rights to grant us this license for any content that you submit to Google Drive.

Dropbox [187]

We collect and use the personal data described above in order to provide you with the Services in a reliable and secure manner. We also collect and use personal data for our legitimate business needs. To the extent we process your personal data for other purposes, we ask for your consent in advance or require that our partners obtain such consent.

Dropbox uses certain trusted third parties (for example, providers of customer support and IT services) to help us provide, improve, protect, and promote our Services. These third parties will access your information only to perform tasks on our behalf in compliance with this Privacy Policy, and we'll remain responsible for their handling of your information per our instructions.