SECURE COMPUTATION IN THE CLOUD USING MAPREDUCE

A thesis submitted to the University of Manchester for the degree of Doctor of Philosophy in the Faculty of Science and Engineering

2018

By
James Dyer
Computer Science
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.3.1</td>
<td>Examples of MR Applications</td>
<td>24</td>
</tr>
<tr>
<td>2.4</td>
<td>Cloud MR Application Scenarios</td>
<td>25</td>
</tr>
<tr>
<td>2.5</td>
<td>Observations from Cloud MR Scenarios</td>
<td>27</td>
</tr>
<tr>
<td>2.6</td>
<td>Additional Challenging Issues for MR in the Cloud</td>
<td>28</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Untrusted Nodes</td>
<td>28</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Dynamic Changes in Cluster Membership</td>
<td>29</td>
</tr>
<tr>
<td>2.7</td>
<td>Concluding Remarks</td>
<td>30</td>
</tr>
<tr>
<td>3</td>
<td>Survey of Related Works and Applicable Methods</td>
<td>31</td>
</tr>
<tr>
<td>3.1</td>
<td>Introduction</td>
<td>31</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Authentication, Authorisation, and Confidentiality</td>
<td>31</td>
</tr>
<tr>
<td>3.1.2</td>
<td>Terminology and Key Concepts from Cryptography</td>
<td>32</td>
</tr>
<tr>
<td>3.2</td>
<td>Applicable Cryptographic Methods</td>
<td>36</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Homomorphic Encryption</td>
<td>36</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Secret Sharing</td>
<td>41</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Order-Preserving Encryption</td>
<td>42</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Searchable Symmetric Encryption</td>
<td>46</td>
</tr>
<tr>
<td>3.3</td>
<td>Related Works</td>
<td>48</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Computing on Encrypted Data</td>
<td>48</td>
</tr>
<tr>
<td>3.3.2</td>
<td>Result Verification</td>
<td>52</td>
</tr>
<tr>
<td>3.3.3</td>
<td>Other Approaches</td>
<td>57</td>
</tr>
<tr>
<td>3.4</td>
<td>Chapter Summary</td>
<td>69</td>
</tr>
<tr>
<td>4</td>
<td>Threat Analysis of MapReduce Computation</td>
<td>70</td>
</tr>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>70</td>
</tr>
<tr>
<td>4.2</td>
<td>A Generic Model of a MapReduce Job Execution</td>
<td>71</td>
</tr>
</tbody>
</table>
10.4 Core Idea .................................................. 205
10.5 Issues ....................................................... 208
  10.5.1 Single Verification Server (VS) ....................... 208
  10.5.2 Input splits containing small numbers of records .... 209
  10.5.3 Malicious Partitioner ................................... 209
  10.5.4 Size of check data ..................................... 209
10.6 High Level Description of the Verification Process ......... 210
10.7 Detailed Description of the Verification Process .......... 211
  10.7.1 Verification of Mapper Results ....................... 211
  10.7.2 Verification of Reducer Results ...................... 213
  10.7.3 Probabilistic Analysis ............................... 215
10.8 Implementation ............................................ 219
10.9 Experimental Results ..................................... 221
10.10 Conclusion ............................................... 228

11 Conclusion ................................. 230

A Implementation of CryptMR .................................. 278
A.1 Introduction .............................................. 278
A.2 CryptMR Encryption Library ................................ 278
  A.2.1 Implemententations of RND and DET Schemes ...... 280
  A.2.2 Implementation of the HOM Scheme .................. 280
  A.2.3 Implementation of the OPE Scheme .................. 280
  A.2.4 Implementation of the SEARCH Scheme ............. 280
A.3 CryptMR Hadoop Writable Library .......................... 281
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B Proofs</td>
<td>289</td>
</tr>
<tr>
<td>B.1 Chapter 7</td>
<td>289</td>
</tr>
<tr>
<td>B.2 Chapter 8</td>
<td>297</td>
</tr>
<tr>
<td>B.3 Chapter 9</td>
<td>298</td>
</tr>
<tr>
<td>C Derivation of Bounds</td>
<td>300</td>
</tr>
<tr>
<td>Glossary</td>
<td>302</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1</td>
<td>Timings for each experimental configuration</td>
<td>162</td>
</tr>
<tr>
<td>7.2</td>
<td>Timings for each experimental configuration</td>
<td>166</td>
</tr>
<tr>
<td>8.1</td>
<td>Timings for each experimental configuration</td>
<td>185</td>
</tr>
<tr>
<td>8.2</td>
<td>Timings for each experimental configuration</td>
<td>189</td>
</tr>
<tr>
<td>9.1</td>
<td>Timings for Evaluated SSE Algorithms</td>
<td>199</td>
</tr>
<tr>
<td>10.1</td>
<td>Meteorological Office Data</td>
<td>204</td>
</tr>
<tr>
<td>10.2</td>
<td>Map phase cheating timings</td>
<td>222</td>
</tr>
<tr>
<td>10.3</td>
<td>Reduce phase cheating timings</td>
<td>225</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Cloud Types .................................................. 17
2.2 Type 1 versus Type 2 hypervisors ....................... 20
2.3 MapReduce data flow ......................................... 22
2.4 MapReduce deployed in a public cloud .................. 26
4.1 Physical architecture of a cloud MapReduce application ........................................ 71
4.2 Software architecture of a cloud MapReduce application ........................................ 73
4.3 Logical architecture of a MapReduce application .................................................. 74
4.4 Worker daemon admission process ....................... 75
4.5 Entity Authentication .......................................... 75
4.6 Job submission process ....................................... 76
4.7 DFS write process in detail .................................. 76
4.8 Worker daemon heartbeat and mapper/reducer launch process ................................ 78
4.9 DFS read process in detail ................................... 78
4.10 Mapper communication processes .......................... 79
4.11 Reducer communication processes .......................... 81
4.12 Job completion notification ................................. 82
7.1 A wire from an input .......................................... 159
7.2 An output wire on a NAND gate ............................. 160
Abstract

Processing large volumes of data has become increasingly important to businesses and government \[218\]. One of the popular tools used in processing large sets of data is MapReduce.

As new MapReduce applications are developed to be deployed in untrusted environments, such as public clouds, or to process sensitive data or to be deployed across data centres, well understood security measures may be deployed, such as authentication and authorisation or encryption of messages passed between cluster nodes. However, there may be situations where authorised individuals cannot be trusted, such as “rogue” system administrators, or, in the cloud, where the MapReduce cluster nodes have been compromised. Where the input data is sensitive, such as medical data, we require a means to protect this data from exposure. Furthermore, we may need to protect the intermediate data and details of the computation from snoopers to prevent information leakage.

To take full advantage of MapReduce cloud computing services, we require a means to process the data securely on such a platform. We designate such a computation, secure computation in the cloud (SCC). SCC should not expose input or output data to any other party, including the cloud service provider. Furthermore, the details of the computation should not allow any other party to deduce its inputs and outputs. Most importantly, we require the computations to be performed in practically reasonable time and space.

The ability to perform MapReduce computations on encrypted data would offer a solution to these problems. However, this poses a significant problem in that many encryption schemes transform the original data in such way that meaningful computation on the encrypted data is impossible.

Our work aims to provide a practical SCC system (CryptMR) inspired by CryptDB \[272\].

Our solution (CryptMR) has detailed several novel cryptographic methods suitable for implementation in a secure computation in the cloud solution using MapReduce. We encrypt integer data using a novel somewhat homomorphic
encryption scheme (SHE). This SHE can be made fully homomorphic. We also provide novel order-preserving (OPE) and searchable symmetric encryption (SSE) for the purposes of sorting and searching. Our OPE scheme is, to our knowledge, the first scheme to rely on a computationally hard primitive rather a security model.

We have implemented all of these encryption schemes and devised experiments to test their suitability for large-scale distributed computing by integrating them into Hadoop MapReduce (MR) applications.

In addition to the work on encryption schemes, we have provided a novel probabilistic method for verifying that mappers and reducers are correctly computing and reporting their results (see chapter 10). This work uses random sampling to minimise the probability that a cheating mapper or reducer can successfully report a false result. We evaluated our “proof-of-concept” implementation in a small scale cloud environment. Our results show that sampling 5 to 10% of intermediate or final data allows us to detect cheating with very strong probability.
Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the “Copyright”) and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made only in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the “Intellectual Property”) and any reproductions of copyright works in the thesis, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=487), in any relevant Thesis restriction declarations deposited in the University Library, The University Library’s regulations (see http://www.manchester.ac.uk/library/aboutus/regulations) and in The University’s policy on presentation of Theses.
Acknowledgements

I would like to thank my supervisor, Professor Jie Xu of the University of Leeds for his guidance and support toward the completion of this thesis. I would also like to thank the Distributed Systems and Services Group and IT Helpdesk at Leeds University for their help and assistance.

I would like to thank my co-supervisor, Professor Norman Paton of the University of Manchester for his support. In particular, I would like to thank him for his suggestion that my MapReduce result verifier could be implemented as MapReduce jobs.

I would like to thank Microsoft for the generous $20,000 Azure for Research sponsorship they provided. This grant enabled me to conduct large-scale experiments that would not have been possible otherwise.

Finally, I would like to thank my family for their support and understanding during this period of study.
Notation

The following notation is used in this thesis:

**Cryptography**
- **Add**: A homomorphic addition function ($\text{Add} : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$)
- **Dec**: A symmetric decryption algorithm ($\text{Dec} : \mathcal{K} \times \mathcal{C} \to \mathcal{M}$)
- **Enc**: A symmetric encryption algorithm ($\text{Enc} : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$)
- **KGen**: A key generation algorithm ($\text{KGen} : \mathcal{S} \to \mathcal{K}$)
- **$\lambda$**: A security parameter
- **MAC**: A message authentication code (MAC) generation algorithm ($\text{MAC} : \mathcal{K} \times \mathcal{M} \to \mathcal{T}$)
- **$\mathcal{C}$**: A set of ciphertext values
- **$\mathcal{K}$**: A set of symmetric secret key values
- **$\mathcal{M}$**: A set of plaintext values
- **$\mathcal{S}$**: A set of security parameter values
- **$\mathcal{T}$**: A set of message authentication code values
- **Mult**: A homomorphic multiplication function ($\text{Mult} : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$)
- **negl($\lambda$)**: A negligible function in $\lambda$, i.e. $< \frac{1}{\text{poly}(\lambda)}$
- **pk**: A public key
- **pk$_X$**: $X$’s asymmetric public key
- **poly($\lambda$)**: Any polynomial in the security parameter $\lambda$
- **sk**: A secret key
- **sk$_X$**: $X$’s asymmetric private key
- **sk$_{X,Y}$**: A symmetric key shared by $X$ and $Y$
- **Vf**: A MAC verification algorithm ($\text{Vf} : \mathcal{K} \times \mathcal{M} \times \mathcal{T} \to \mathbb{B}$)
- **$c, c_1, c_2, \ldots$**: Ciphertext values
- **$m, m_1, m_2, \ldots$**: Plaintext values
with high probability with probability $1 - 2^{-\epsilon \lambda}$, for some constant $\epsilon > 0$

**Mathematics**

$(x, y) \quad$ The integers between $x$ and $y$ exclusive

$[x, y) \quad$ The integers between $x$ and $y$ including $x$ but excluding $y$

$[x, y] \quad$ The integers between $x$ and $y$ inclusive

$[x] \quad$ The largest integer less than or equal to $x$ (the “floor” operator)

$\mathbb{B} \quad$ The set of Boolean values ($\text{True}$ and $\text{False}$)

$\mathbb{N} \quad$ The set comprising zero and the positive integers

$\mathbb{R} \quad$ The set of real numbers

$\mathbb{R}[x, y) \quad$ The real numbers in the interval $[x, y)$

$\mathbb{Z} \quad$ The set of positive and negative integers

$\mathcal{S}^n \quad$ The space of $n$ element tuples where each element of the tuple is a member of the set $\mathcal{S}$

$e_i \quad$ The $i$th unit vector ($i = 1, 2, \ldots$), with size determined by context

$v_* \quad$ A $k^*$-vector ($k^* = \begin{pmatrix} k+1 \\ 2 \end{pmatrix}$), $[v_1 \ v_2 \ \ldots \ v_{k^*}]^T$, which augments the $k$-vector $v = [v_1 \ v_2 \ \ldots \ v_k]^T$ by appending elements $v_i = f_i(v_1, \ldots, v_k)$ ($i \in [k+1, k^*]$), for a linear function $f_i$

$\Pr[P|Q] \quad$ The probability that predicate $P$ is true given that predicate $Q$ is true

$\Pr[P] \quad$ The probability that predicate $P$ is true

$\mathbb{Z}_n \quad$ The finite field of integers modulo $n$

$x \overset{p}{\in} \mathcal{S} \quad$ $x$ is a prime chosen uniformly at random from the discrete set $\mathcal{S}$

$x \overset{s}{\in} \mathcal{S} \quad$ $x$ is chosen uniformly at random from the discrete set $\mathcal{S}$

**Security Protocols**

$ID_X \quad$ Data that identifies $X$, such as a username

$X \rightarrow Y : Z \quad$ $X$ sends message $Z$ to $Y$

**Symbols**

$\perp \quad$ The logical constant “bottom”, used here to represent an inconsistent or error state

$\in \quad$ Membership of a set
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lg</td>
<td>Logarithm to the base 2</td>
</tr>
<tr>
<td>ln</td>
<td>Logarithm to the base $e$</td>
</tr>
<tr>
<td>log</td>
<td>Logarithm to an unspecified base</td>
</tr>
<tr>
<td>⊕</td>
<td>Exclusive or (XOR) operator</td>
</tr>
<tr>
<td>∥</td>
<td>Concatenation operator</td>
</tr>
<tr>
<td>[]</td>
<td>The empty list</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Introduction

Where processing large volumes of data was once the province of academia and specialised applications, it has become increasingly important to other areas such as business and medical research [218]. One of the popular tools used in processing large sets of data is MapReduce. Introduced in 2004, in their OSDI conference paper [105], Dean and Ghemawat described Google’s MapReduce parallel programming model used in many internal Google applications, along with its associated implementation. This paper has inspired a number of similar technologies, such as the open-source Hadoop MapReduce [17]. MapReduce is currently used in approximately 80% of Google’s data processing computations [363], and Hadoop MapReduce forms the basis of many tasks at Yahoo! and Facebook [58].

MapReduce is a data parallel model of distributed computing. Data to be processed is divided between worker programs. Its popularity is largely because application parallelism is handled by the MapReduce application framework. The user-defined contribution to a MapReduce application simply defines the operations to be performed on data by a worker process. This simplified programming model makes it an attractive framework on which to develop data parallel applications.

Cloud computing has made MapReduce more accessible. Infrastructure services (Infrastructure-as-a-Service) provided by cloud vendors allow easy provisioning of
large numbers of compute instances. MapReduce platform services (Platform-as-a-Service) are provided by cloud vendors such as Amazon and Microsoft. MapReduce clusters can now be built for reduced cost or paid for on a per use basis.

For many MapReduce applications, performance has been paramount and security has taken a back seat. Network isolation is the sole security control applied to many MapReduce applications. As new MapReduce applications are developed to be deployed in untrusted environments, such as public clouds, or to process sensitive data or to be deployed across data centres, network isolation is not sufficient to provide adequate security. Furthermore, sophisticated authentication and authorisation measures cannot prevent a “rogue” system administrator from accessing confidential data. Therefore, a means of making MapReduce computations private is required. By this, we mean that even privileged individuals, such as system administrators, will be unable to “snoop” on the data that the MapReduce computation consumes and produces.

This thesis presents the results of work undertaken toward providing a method for secure computation in the cloud, primarily using MapReduce. Chapter 1 provides an introduction to the research problem and details the motivation for this research. Chapter 2 provides explanations of cloud computing and MapReduce. In addition, it details the additional challenges to security posed by deploying MapReduce applications in the cloud. Chapter 3 presents a survey of relevant cryptographic literature. We conclude the literature review with a survey of approaches to security for MapReduce. Chapter 4 presents a generic model of MapReduce computation. Furthermore, this chapter presents a threat analysis of MapReduce computations based on the generic model. Following the conclusion of this chapter that computing over encrypted data is the best way forward, chapter 5 discusses cryptographic security models used in provable security, in particular those referred to in later chapters. Chapter 6 introduces our secure single-party computation in the cloud (SCCC) system which leverages MapReduce (CryptMR). Chapter 7 details the novel homomorphic encryption schemes over the integers which are employed in CryptMR for arithmetic computation. Chapter 8 details the novel order-preserving encryption scheme used in CryptMR for sorting data. Chapter 9 details the novel searchable symmetric encryption scheme which CryptMR uses to support searching. Chapter 10 presents our work on a result verification system for MapReduce. Finally, chapter 11 concludes the
1.2. IMPLICATIONS OF SECURITY THREATS TO MAPREDUCE

This thesis also contains three appendices. In appendix A we present details of the implementations of CryptMR and the experiments used to evaluate its performance. In appendix B we provide proofs of all theorems and lemmas in this thesis. Finally, in appendix C we provide the derivation of the bounds on the security parameters discussed in section 7.2.1.

1.2 Implications of Security Threats to MapReduce

As the demand for applications that are capable of processing large quantities of data has grown, MapReduce has become an increasingly popular programming model and software architecture. Not only is MapReduce employed in academia, it is the cornerstone of Google, Yahoo! and Facebook’s data processing software[58, 363]. Furthermore, many new information processing start-ups are using Apache Hadoop as the basis of their software [160].

These large volume data (“big data”) processing applications are now handling sensitive information. Facebook, Google and Yahoo! process the personal data of their users to better target advertising. In New York, patient data was analysed to better target medical care[134]. ANZ Bank is using “big data” processing to better target customer needs [317]. Information leakage of patient medical data or bank customer financial data would prove highly damaging for the reputation of the organisation and, possibly, devastating for the individuals whose data has been compromised.

If MapReduce is to be used as a tool for processing sensitive “big data” then it requires a security framework which makes information leaks, such as those above, unlikely. However, common MapReduce run-times either provide no security framework, or an inadequate one as we discuss in section 3.3.3.
1.3 Research Motivation and Challenges

As new MapReduce applications are developed to be deployed in untrusted environments, such as public clouds, or to process sensitive data, or to be deployed across data centres, well understood security measures may be employed, such as authentication and authorisation or encryption of messages passed between cluster nodes. However, there may be situations where authorised individuals cannot be trusted. We may have “rogue” system administrators, who wish to snoop on the data, or the MapReduce cluster nodes may have become compromised. Where the input data is sensitive, such as medical data, we require a means to protect this data from exposure. Furthermore, we may need to protect the intermediate data and details of the computation from snoopers to prevent information leakage.

In addition, a cloud MapReduce application poses additional challenges. The application is deployed in an untrusted environment. Unlike an application wholly managed by an organisation, the nature of deployment in the cloud means that some workers cannot be trusted. An ideal security solution needs to address the possibility that workers may be colluding to disrupt the computation. Furthermore, a cloud MapReduce application may be rapidly scaled up or down. The ideal security solution should be able to handle the admission and ejection of cluster nodes. These additional challenges are discussed in detail in section 2.6.

These final three requirements: performance, untrusted workers, and dynamic changes in cluster membership; make providing a security solution for MapReduce using security measures, such as authentication and authorisation, a very challenging problem.

The ability to perform MapReduce computations on encrypted data would offer a solution to most of these problems. However, this poses a significant problem in that many encryption schemes transform the original data in such way that meaningful computation on the encrypted data is impossible. However, recent groundbreaking advances in fully homomorphic encryption (FHE) \[137\] offer the potential for computing functions on encrypted data and thereby securing computation in the cloud \[88\][308]. This encryption system allows a function, represented as an arithmetised circuit \[23\], to be computed without decrypting the ciphertext. As it is a homomorphism, the result of computing the function
on the ciphertext is the same as encrypting the result of computing the function on the plaintext. However, as currently proposed, FHE schemes are currently highly time and space inefficient compared to computation on unencrypted data (see \[3.2.1\]). In conclusion, fully homomorphic encryption seems impractical in its current form. What is required is a practical solution that allows MapReduce to be compute over encrypted data.

1.4 Novel Contributions

The following are the contributions of this thesis:

Threat analysis We have conducted a thorough threat analysis of MapReduce communication. This material is presented in chapter \[4\].

Novel encryption schemes We have detailed several novel cryptographic methods suitable for implementation in a secure computation in the cloud solution using MapReduce. We should also note that the schemes we have devised are transferrable to a wide variety of distributed and non-distributed computing paradigms and are not restricted to MapReduce.

Homomorphic encryption scheme for integer data We encrypt integer data using a family of novel *somewhat homomorphic encryption* (SHE) schemes (chapter \[7\]). We also show that our generalised SHE can be made fully homomorphic. Although this is an active research area, we believe that our approach offers significant improvements over related work \[272\] \[311\] in that previously implemented related work encrypts integer data according to which arithmetic operation (addition or multiplication) is to be performed on it. Under such a scheme, for an operation which is the combination of several additions and multiplications, such as an inner product, the data must be decrypted and re-encrypted after every addition or multiplication operation before the computation can proceed. With regard to private MapReduce computations in the cloud this would involve shipping the intermediate operation results back to a trusted client to decrypt and re-encrypt the data. This would create a significant overhead in
execution time. Our scheme does not require this, as additions and multiplications could be performed on the same encrypted integer data.

Order-preserving encryption (OPE) scheme Our OPE scheme allows for sorting of encrypted data. It is, to our knowledge, the first scheme to rely on a computationally hard primitive rather a security model (chapter 8).

Searchable symmetric encryption (SSE) scheme Our SSE scheme allows for searching on encrypted data. It is based on Song et al.’s [309] work but addresses some of the impracticalities of that scheme (chapter 9).

MapReduce result verifier In addition to the work on encryption schemes, we have provided a novel probabilistic method for verifying that mappers and reducers are correctly computing and reporting their results (see chapter 10). This work uses random sampling to minimise the probability that a cheating mapper or reducer can successfully report a false result.
Chapter 2

Problem Background

2.1 Introduction

In this chapter, we review background concepts relating to cloud MapReduce (MR) applications. We review key terms and concepts relating to cloud computing and MR. We first outline the key concepts and terminology of cloud computing. We then provide a high-level overview of the MR programming paradigm and software architecture. Finally, we discuss how a MR application is deployed in a cloud environment and what challenges this poses to security.

2.2 Cloud Computing

As we are interested in MR applications deployed in cloud environments, this section will provide a brief review of cloud computing and define the terminology which we will refer to in this report.

Cloud computing is intended to deliver computing as a service. Cloud services are delivered over a network, usually the Internet, and are provided as a utility. The term “cloud” derives from computer networking where a collection of undefined resources, for example, Internet routers, switches, and hosts, are represented as a cloud. With regard to cloud computing, the “cloud” is the collection of undefined resources, invisible to the consumer, that are required to provide a cloud service. Cloud computing exhibits the following key characteristics:
On-demand self-service Cloud services are automatically provisioned and de-provisioned at user request;

Ubiquitous network access Cloud services are accessed using standard network and application protocols;

Resource pooling Use of virtualisation technologies allows physical resource to be shared thereby improving utilisation and reducing capital cost. Cloud applications share tenancy on physical hosts with other resources. In addition, cloud applications can easily be migrated from one physical host to another, improving application availability;

Rapid elasticity Resources can be quickly scaled up or down;

Measured service Computing resources, such as storage or computation, are packaged and provided as a metered service (like the electricity grid).

2.2.1 Cloud Taxonomy

Cloud computing services have been divided into three broad categories which are:

Software as a Service (Saas) Software-as-a-service (SaaS) applications, such as Google Docs or Hotmail are analogues of user desktop applications. The user interface to these web applications is intended to mirror similar interfaces on user desktop applications;

Platform as a Service (PaaS) Platform-as-a-service (PaaS) is intended to be an analogue for the operating system. As an operating system vendor supplies APIs and middleware to application developers enabling them build applications for that environment, the cloud provider supplies cloud services and middleware which can be used to rapidly build and deploy hosted customer applications. Although some hosting details are decided by the customer, the infrastructure implementation of a service by the cloud provider is largely opaque to the customer. With regard to MR, Amazon’s Elastic MapReduce service [10] and Microsoft’s Hadoop on Azure[227] are examples of MR PaaS implementations;
2.2. CLOUD COMPUTING

Infrastructure as a Service (IaaS) Infrastructure-as-a-service (IaaS) is the cloud analogue for physical hosts. This model provides the most flexibility but requires a substantial manpower component with regard to deploying and maintaining the infrastructure. The cloud provider offers virtual infrastructure resources such as virtual switches and hosts. Although operating systems are pre-installed on virtual hosts, a customer must configure their own network topology and install additional required application software. With regard to MR, the majority of cloud MR implementations fall into this category. The customer would build their own MR cluster using virtual hosts and virtual switches and install the MR framework on the cluster nodes.

![Cloud Types Diagram](image)

Figure 2.1: Cloud Types

We define four types of cloud environment, three of which are displayed in figure 2.1. Each type designates where the administrative control for the cloud resources lies. These four types are:

**Private cloud** A private cloud provides cloud services within an organisation. It is hosted and managed on an organisation’s own resources and is isolated from the Internet. Private clouds are typically implemented using virtualisation software such as VMware’s vSphere 324, Citrix’s Xen 90 or KVM 280 and cloud platform software such as OpenNebula 255 or OpenStack 256;
Public cloud A public cloud provides cloud services to the Internet. Public clouds are hosted and managed by cloud providers such as Amazon, Google, and Microsoft. Public clouds are implemented using internal proprietary cloud platform software;

Hybrid cloud A hybrid cloud provides cloud services within an organisation using a combination of private and public cloud resources;

Community cloud A community cloud is a cloud shared between several organisations from the same community. It can be managed by these organisations or by a third party and may be hosted internally or externally.

2.2.2 Virtualisation

As stated in section 2.2 cloud computing platforms rely on virtualisation technologies. In this section we describe what is meant by virtualisation, review core concepts and define key terminology.

Virtualisation is best described by comparing and contrasting it with two similar technologies: emulation and simulation. In emulation, a software layer provides a representation of physical hardware. Machine instructions provided by an operating system running above the emulation layer are interpreted by the emulation layer as native instructions for the physical hardware the emulation layer runs on. Emulation allows software compiled for one physical architecture to be executed on a differing physical architecture without being rebuilt for the new architecture. However, emulation typically has a large processing overhead and emulated applications can see severely degraded performance. An example of emulation is PearPC, a PowerPC emulator. In simulation, a software layer presents objects that appear, to applications above the simulation layer, to behave as native objects. However, the underlying implementation of such objects may be radically different from those on the native platform. An example of a simulated environment is Linux’s WINE, which presents an implementation of the Windows API in Linux. This API allows native Windows applications to run on top of Linux. However, the implementation of the API calls is different from the native Windows platform. Virtualisation shares characteristics of both emulation and simulation. The virtualisation software layer presents software objects,
virtual hardware, to a guest operating system which behave like physical devices. Some of these virtual hardware components are simulated, that is, they are represented entirely in software, some are emulated, that is, instructions from the guest operating system are passed to the physical devices by the virtualisation layer. In the case of some components, such as RAM, the virtualisation layer presents a mapping from a virtual object, a page of virtual RAM, to physical resources, a physical page. The processing overhead for virtualisation is not as severe as emulation, largely because the virtual hardware must conform to the same architecture, e.g. Intel x86, as the physical hardware, and it does not require interpretation of instructions. Therefore, operating systems running on virtual hardware must also conform to this architecture.

The virtualisation layer is capable of presenting and managing a number of collections of virtual hardware components, called virtual machines (VMs), so that they can share resources on the physical host. This sharing of resources allows for greater utilisation of physical resources. This results in a multi-tenant environment where several guest operating systems reside on the same physical host. It should be noted that, for a particular piece of virtualisation software, regardless of the underlying physical hardware components, the virtual hardware components are the same for each VM. This means that VMs can be specified as a configuration file along with files for persistent virtual devices, such as disks. Provision of a VM can be a simple copy from a set of template files or creation of new set of files. The VM can be decommissioned by deleting these files. Furthermore, a VM can be easily moved from one physical host to another by copying these files. This enables physical virtualisation hosts to be clustered to enable load balancing of resources or power management.

The virtualisation software layer is called a hypervisor or virtual machine monitor (VMM). A hypervisor which runs on top of a host operating system is designated as a type 2 hypervisor. Examples of type 2 hypervisors include VirtualBox \[258\], QEMU \[275\], and Parallels Desktop \[263\]. A hypervisor which runs directly on top of physical hardware is a type 1 hypervisor. Figure 2.2 illustrates this. A type 1 hypervisor is, in essence, a highly specialised operating system whose sole purpose is to manage VMs. In addition, to providing virtual hardware for guest operating systems, type 1 hypervisors also provide software switches to network VMs. Examples of type 1 hypervisors include Citrix XenServer \[90\], Microsoft
Cloud computing environments use virtualisation because it provides a platform where computing resources can be fully utilised and units of computations can be rapidly created, deployed and destroyed. This offers the scalability and flexibility demands that cloud computing requires. In order to offer self provision, a cloud platform adds middleware that allow users to create and launch VMs themselves. A VM provisioned in a cloud environment is called a compute instance. Cloud providers typically use clusters of type 1 hypervisors to provide computing resources. The VMs running on these clusters are used to provide internal and external cloud services.

2.2.3 Containerisation

A major issue with virtualisation is the processing overhead required, along with the expenditure of resources, such as RAM and file space, required to represent the VM. Therefore, some cloud providers do not isolate customer applications using VMs but rather by application containerisation or operating-system-level virtualisation. Containerisation is a means of packaging an application so that it will execute in its own sandbox on the host server. In particular, application containers use the host’s operating system call interface rather than emulation as is the case in virtualisation. Like VMs, containers can be rapidly deployed and, in some implementations, may be migrated between hosts with minimal loss of service.

Containers are seen as a lightweight alternative to virtualisation. Implementations include the multi-platform Docker [113], Linux-vServer [135], and Sandboxie for Windows [291], among many others.
2.3 MapReduce

In this section we present an overview of the MR application model [105] and define key concepts relating to MR. The MR application model fundamentally depends on two user-defined subroutines, Map and Reduce. The Map subroutine applies the same operation to each individual input record, whereas the Reduce subroutine combines the outputs of Map subroutines into some desired resulting set of records. The design of these two subroutines is derived from functional programming. In functional programming, there exist two higher order functions, map and fold, which are defined as:

**Definition 1** (Map function).

$$\text{map}(f, l : \text{list}) = \begin{cases} \[] & \text{if } l = \[] \\ f(x) \parallel \text{map}(f, l') & \text{if } l = x \parallel l' \end{cases}$$

**Definition 2** (Fold function).

$$\text{fold}(f, z, l : \text{list}) = \begin{cases} z & \text{if } l = \[] \\ \text{fold}(f, f(z, x), l') & \text{if } l = x \parallel l' \end{cases}$$

For example, $\text{map}(\text{square}, [1, 2, 3, 4, 5]) = [1, 4, 9, 16, 25]$, where $\text{square}(x) = x^2$, or $\text{fold}(\text{sum}, 0, [1, 2, 3, 4, 5]) = 15$, where $\text{sum}(x, y) = x + y$.

If we are to think of the list inputs to map and fold as lists of records, then we obtain the inspiration for the design of the Map and Reduce subroutines of MR. Map is applied independently to every input record. Therefore it can be easily parallelised by dividing the input records by the number of workers and assigning each worker a subset of the data. Depending on the associativity and commutativity of the Reduce subroutine, some portions of the Reduce phase of computation might also be parallelised.

A high level overview of the components and data flow of MR is shown in Figure 2.3. An individual MR application is called a job. A job is composed of a number of job related programs called tasks. The middleware and APIs that support a MR job are the MR application framework. Parallelism in MR is achieved by dividing the input data amongst the tasks. Each task will be executed by a parallel worker process. In the classic MR application model, we have two tasks, map and reduce, and the input data is divided into a number of groups equal to
Input data
Map
Map
Map
Reduce
Reduce
Reduce
Reduce
Output
Output
out
out
out
out
Split
1
Split
2
Split
3
Intermediate data
Key-value pairs
Key-value pairs
Key-value pairs
parse
parse
parse
in
in
in
out
out
out

Figure 2.3: MapReduce data flow (adapted from [105])

The number of map tasks. Each map task is assigned a division of the data called an input split which it processes, outputting intermediate data. The intermediate data is divided into a number of partitions, where there is one partition for each reduce task. Each reduce task will receive the same indexed partition of intermediate data from each map task. A typical MR program will have fewer reduce tasks than map tasks.

The majority of subroutines in a task are pre-defined by the framework. Two subroutines, Map and Reduce, are user-defined. Map and Reduce are side effect free functions called by the framework as part of the map and reduce task respectively. The fact that the amount of user-defined code is small in comparison to the framework code is a major attraction of the MR application model. Map and Reduce are, typically, sequential functions. In the majority of MR applications, Map and Reduce operate on (key, value) pairs. This is because many MR applications process semi-structured data which is to be stored in non-relational data stores. Input must be parsed into pairs before the map task can apply the Map function to each pair.

MR has three phases of computation: the map phase, the shuffle phase, and the reduce phase. During the map phase, the map tasks execute. Each map task parses the input split into (key, value) pairs and applies the Map function to each pair. The output of each application of the Map function is stored locally as intermediate data and partitioned. The shuffle phase collects the equally indexed partitions of intermediate data from every map task, collates the data, and distributes each collated partition to the reduce task to which it is assigned. Additionally, during the shuffle phase, each set of pairs in the partition is sorted and grouped by key. Grouped pairs are merged into a single (key, list of values)
pair. In the reduce phase, the reduce tasks execute. Each reduce task receives a partition of the intermediate data. The reduce task applies the \textit{Reduce} function to each \textit{(key, value)} pair in the partition and writes the output data to global storage.

The function signature for \textit{Map} is as follows:

\textbf{Definition 3} (Map function signature). \textit{Map}(k_1, v_1) = [(k_2, v_2)]

The signature states that \textit{Map} takes a \textit{(key, value)} pair as input and emits a list of \textit{(key, value)} pairs as output, where \( k_1 \in K_1 \) (a set of keys of type \( \alpha_1 \)), \( v_1 \in V_1 \) (a set of values of type \( \alpha_2 \)), \( k_2 \in K_2 \) (a set of keys of type \( \beta_1 \)), and \( v_2 \in V_2 \) (a set of values of type \( \beta_2 \)). The pseudocode (algorithm 2.1) below provides an example \textit{Map} function for a distributed word count application:

\begin{algorithm}
\begin{algorithmic}[1]
\State \textbf{Input} : key: document name
\State \textbf{Input} : value: document contents
\State \textbf{Output}: result: list of key-value pairs
\State \texttt{result} ← []
\State \texttt{forall } w \in \texttt{value} \texttt{ do}
\State \hspace{1em} \texttt{result} ← \texttt{result} ∥ (w, 1) \hfill // w : a word in document
\State \texttt{end}
\State \texttt{return \hspace{1em} result}
\end{algorithmic}
\caption{Example map function for distributed word count}
\end{algorithm}

The function signature for \textit{Reduce} is:

\textbf{Definition 4} (Reduce function signature). \textit{Reduce}(k_2, [v_2]) = [v_3]

The signature states that \textit{Reduce} takes a pair of \textit{(key, list of values)} and emits a list of values, where \( k_2 \in K_2 \) (a set of keys of type \( \beta_1 \)), \( v_2 \in V_2 \) (a set of values of type \( \beta_2 \)), and \( v_3 \in V_3 \) (a set of values of type \( \gamma \)). Algorithm 2.2 provides the \textit{Reduce} function for our example word count application.

MR application frameworks share some characteristics. These characteristics are as follows.

\textbf{Data parallel} MR style applications achieve parallelism by dividing the input data among the workers so that each worker can apply operations to its subset of data.
Algorithm 2.2: Example reduce function for distributed word count

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>key: word</td>
<td>(key, result): pair of word and number of occurrences</td>
</tr>
<tr>
<td>values: a list of ones</td>
<td></td>
</tr>
</tbody>
</table>

1. result ← 0
2. forall \( v \in \text{values} \) do
3. \( \text{result} \leftarrow \text{result} + v \)
4. end
5. return (key, result)

Shared nothing Each worker completes its computations based on initial data assigned to it and does not require additional input from other workers. Furthermore, if a worker fails, its failure does not effect other workers. However, the data computed by that worker is lost and the work assigned to it must be recomputed.

Side effect free Execution of a worker does not affect the behaviour of execution of other workers.

Loosely coupled The MR application framework is designed for clusters that can have high latency between nodes. Therefore, subroutines executed at workers are loosely coupled to prevent bottlenecks.

RAIN (redundant array of inexpensive nodes) The MR application framework was designed to be deployed on large clusters of commodity machines. Within a cloud computing environment each cluster node will likely be a compute instance. The framework is designed so that the physical architecture of individual cluster nodes is not important.

These characteristics mean that MR applications can be scaled easily by adjusting the number of workers. Furthermore, the shared nothing architecture means that if a worker fails, its work can be reassigned to another worker allowing the computation to complete. This makes MR highly resistant to worker failure.

2.3.1 Examples of MR Applications

MR applications typically fall into three areas: applications that process text, particularly the tokenization, indexing and searching of text; applications that
create new data structures, such as graphs; and data mining and machine learning applications.

In this section we present a small selection of MR applications to demonstrate how the framework is used.

**Distributed Grep**  The Unix grep function searches for occurrences of a regular expression in a document. Distributed grep applies a similar regular expression search to large collections of documents. As a MR application, the map subroutine emits a line from a document if it matches the regular expression. The reduce subroutine is an identity which simply copies the intermediate data to the output files.

**Distributed Sort**  MR has been used to distributively sort large volumes of data. Hadoop’s algorithm, TeraSort [250], sorts the input for each input split in the map phase. A custom partitioner in the map subroutine guarantees that data sent to each reducer is sorted with respect to all other reducers.

**Count of URL access frequency**  This MR application processes web page request logs. The map subroutine outputs pairs of \((URL, 1)\). The reduce subroutine aggregates all values for the same URL to output pairs of \((URL, \text{total number})\).

**Web-link graph reversal**  This MR application processes web documents and finds which source URLs contain links to a target URL. The map subroutine outputs \((\text{target}, \text{source})\) pairs. The reduce subroutine concatenates the the pairs by target URL to produce an output pair \((\text{target}, \text{list of source URLs})\).

### 2.4 Cloud MR Application Scenarios

MR applications may be deployed in a public cloud in one of two ways. The first method is where the MR application framework is offered as a platform-as-a-service (PaaS), as with Amazon Elastic MapReduce [10] and HDInsight [227]. In this case, the user submits an application to the cloud management interface and also selects the number of pre-configured compute instances to use. These instances have the MR application framework installed on them. When the user
start the MR job, the PaaS middleware launches the compute instances and the framework daemons and then starts the job. Other than the selection of how many compute instances to use, the cloud implementation of the MR application is hidden from the end user. Furthermore, the only interaction the end user has with the MR application framework and the compute instances is through the web-based management interface.

The second method deploys the MR application on infrastructure-as-a-service (IaaS). In this case, the end user builds a virtual cluster using a generic compute instance. This generic compute instance, which has an uncustomised operating system installed, is configured by the end user. Part of this configuration is the installation of the MR framework daemons. Compute instances have a public IP address and are configured and managed remotely by the end user using software such as SSH. Once configured, the customised compute instance is cloned to provide as many compute instances as required. The compute instances are launched and the MR framework daemons are started on each instance. At this point, the MR cluster is operational. The end user logs in to the master node remotely to configure and start the job. An IaaS deployment requires significant interaction with the cluster compute instances and with the MR framework daemons. However, it also allows the MR framework to be customised to the end user’s requirements.

In either case, the MR application is physically deployed as VMs in the public cloud (see Figure 2.4). One VM is the MR master node and the other VMs are
2.5 Observations from Cloud MR Scenarios

Section 2.4 outlines typical deployment scenarios for cloud MR applications. From these scenarios we can make a number of observations.

**Loss of administrative control** Cloud services require that customers relinquish administrative control of their application to a greater or lesser degree. The administrative control of a MR application in a cloud is shared between the application owner and the cloud provider. The customer trusts that the cloud provider has trustworthy system administrators, has provided network security controls, kept compute instances updated, and stored data securely. With regard to an untrustworthy system administrator, such an individual has considerable privilege to “snoop” on or alter the customer’s data or to disrupt the MR computation. Regarding data, there can exist...
policies as to where data should be stored, transmitted and used. If this data is stored in the cloud these policies may be ignored, particularly where the cloud provider is located in a different geographical location from the data owner or where the cloud provider wishes to monetarise data. Furthermore, the management and maintenance of the virtualisation hosts that compute instances run on is also performed by the cloud provider. An unpatched or misconfigured server may be exposed to attack. As discussed above, a compromised hypervisor allows an attacker to gain control of the virtual machines running on it. Similarly, the cloud middleware is third party software and flaws in this middleware may allow unauthenticated access to the customer application.

**Opaque physical architecture** The physical deployment of the virtual machines that comprise an application is opaque. The customer has no knowledge of the cloud architecture and is unable to determine the physical location of virtual machines within this architecture.

**Shared resources** Application VMs share physical computing and network resources with other applications. These shared resources may be used against an application by an attacker. Applications are only isolated in so far as the cloud provider safely partitions these resources.

### 2.6 Additional Challenging Issues for MR in the Cloud

In this section, we outline the features of a cloud MR application which provide additional challenges for security.

#### 2.6.1 Untrusted Nodes

As a result of the use of virtualisation, cluster nodes in a cloud MR application may not be trustworthy. The customer software running on the virtual machines (see Figure 2.4) is only isolated from other customer applications in so far as the virtualisation host’s hypervisor is secure. If the hypervisor is compromised then
all virtual machines running on the host can be controlled by an attacker. A hypervisor presents an additional attack surface for the virtualisation host. The hypervisor management interface exposes it to network attacks. Similarly, the software switches and simulated hardware drivers, both of which interact with the hypervisor, expose it to side channel attacks using malicious virtual machines running on the host. Furthermore, two popular hypervisors, Xen and Hyper-V, use a highly privileged virtual machine to manage physical hardware access and control the hypervisor. This VM is known as "Dom0" for Xen and as the "Parent Partition" for Hyper-V. Compromising this VM is equivalent to compromising the hypervisor itself. It should be noted that the Xen hypervisor is used in many cloud implementations, such as Amazon’s Elastic Compute Cloud (EC2) and the Hyper-V hypervisor is used as the basis for Microsoft’s Azure platform. In addition to the threat posed by the hypervisor, an infrastructure-as-a-service provider allows customers to configure VMs as they wish. A customer is able to install an unpatched operating system on a VM. In addition, the cloud provider makes a repository of pre-configured VMs available for customers to use. Customers are also able to add VMs to the repository. An attacker is able to upload a trojanised VM to the repository which may then be used as part of a customer application. Finally, cloud VMs have public IP addresses by default which make it possible for an attacker to “sniff” on ports to discover and exploit vulnerabilities.

Therefore, it is quite possible that nodes in the cluster will be working alone or in collusion to subvert the MR computation. Such nodes may be attempting to steal data from the application or to sabotage the computation.

2.6.2 Dynamic Changes in Cluster Membership

One of the attractions of cloud computing is rapid elasticity, that is, the ability to quickly scale up or down an application. With regard to a cloud MR application, this is achieved by adding or removing compute instances to and from the MR cluster. If we are able to add cluster nodes dynamically as the computation executes then there is a possibility that an attacker’s VM could try to gain admittance to the cluster. Alternatively, cloud middleware designed to control the shutdown of VMs could be exploited by an attacker to maliciously remove
resources from the cluster.

2.7 Concluding Remarks

This chapter has reviewed key concepts of cloud computing and MR. It has then focused on MR applications deployed in a cloud environment. From this focus, we identify a number of key characteristics of cloud MR applications. These characteristics are as follows.

**Tolerant of worker failure** If a worker fails, rescheduling its work to another worker is all that is necessary to complete the computation.

**Highly and elastically scalable** The MR paradigm makes it easy to scale applications by adding workers. When deployed in the cloud, an application can be scaled dynamically as the application executes.

**Untrusted workers** Workers in a cloud MR application may be working in isolation or in collusion to disrupt the computation.

**Loss of administrative control** Large portions of a cloud MR cluster may be in the administrative control of another, possibly malicious, entity.
Chapter 3

Survey of Related Works and Applicable Methods

3.1 Introduction

In this chapter, we review security methods which are applicable to the problem space. Methods which are pertinent to MapReduce are discussed in detail. In addition, we review related work.

3.1.1 Authentication, Authorisation, and Confidentiality

As many of the security methods discussed in this chapter discuss authentication, authorisation, and confidentiality, we define these terms here.

Authentication is the process of proving the identity of something, be it an individual or a message. For example, a person is challenged by a security guard at a checkpoint to establish their identity. The person would produce a trusted piece of information, such as identity papers or an identity card, which establishes their identity. We call this trusted piece of information a credential. Furthermore, the challenge–response process described is an example of an authentication protocol. An authentication protocol is a process understood by both claimant, the entity wishing to authenticate, and verifier, the entity verifying the identity, by which the identity of the claimant may be verified.
We divide authentication into two particular usage scenarios. The first is peer-entity authentication which is used to authenticate the actors in a particular interaction. The second is message authentication which seeks to establish whether a received message is exactly the same as that sent and that the purported sender identity is valid.

The way in which an entity may be authenticated is known as an authentication factor. If an authentication method combines several different factors we call this multi-factor authentication. Examples of factors include password, identity cards, security tokens and biometric data such as fingerprints or iris patterns.

Authorisation or access control is the process by which an authenticated entity is certified to perform an action. Using our real world example, presenting an identity card to the guard may elicit the response, “Thank you, but you are not authorised to enter this area,” depending on the security clearance of the individual. The set of actions that may be performed, the set of entities that can perform them, and the mapping between these sets that dictates which entities are allowed to or denied from performing an action, forms an authorisation policy.

Confidentiality is the process by which information is stored and exchanged so that only authorised entities may be able to access it. As a real-world example, a letter marked “Management Only” is not confidential unless it is placed in an sealed envelope and addressed to an authorised individual. Confidentiality of data is usually achieved by cryptographic methods.

3.1.2 Terminology and Key Concepts from Cryptography

As many of the security methods reviewed in this chapter are based on cryptographic methods, we briefly review the key concepts and terminology of cryptography here. Cryptography is the method of making data private by applying a function to readable input, known as plaintext, which renders a disguised output, known as ciphertext. This process is called encryption or enciphering. The ciphertext may then be freely transmitted, particularly across insecure channels such as the Internet, to its intended destination. If the ciphertext is intercepted, it should not be possible for the intercepter to reverse the encryption process and retrieve the plaintext. However, the intended recipient is able to compute the inverse of the encryption function and retrieve the plaintext. This process is
known as decryption or deciphering.

Encryption and decryption functions typically require the input of a secret parameter known as a key. If the key must be shared by the encrypting and decrypting parties, then we call this symmetric cryptography. If, however, there are two keys, one known by the enciphering party and another known by the deciphering party, then we call this asymmetric cryptography. The encryption and decryption algorithms along with all possible input and key values forms a cryptosystem.

Symmetric cryptographic methods typically use complicated processes to disguise the plaintext. A symmetric cipher is a trio of functions, one for encryption, one for decryption, and one for generating the shared secret key. The key generation function generates a secret key according to some security parameter, usually the key length in bits. The encryption function, Enc, takes two inputs, the shared secret key, $k$, and the plaintext, $m$, to produce ciphertext, $c = \text{Enc}(k, m)$. Similarly, the decryption function, Dec, takes two inputs, the shared secret key, $k$, and the ciphertext, $c$, to retrieve the plaintext, $m = \text{Dec}(k, c)$. Common symmetric ciphers, such as Data Encryption Standard (DES) and Advanced Encryption Standard (AES), operate on several bytes of input, a block, and are called block ciphers. The common components of a block cipher are substitution, exchanging a byte of plaintext with a different byte, permutation, shifting the bytes of data around within a block, and adding the key material by an exclusive or (XOR) operation. A “good” block cipher produces output which looks like random data. Therefore, block ciphers are composed of many of these component operations to ensure that the ciphertext is considered sufficiently indistinguishable from a random sequence.

Common asymmetric encryption methods are public key encryption schemes. In this scheme, one key is known only by an individual, the private key, whereas the other is freely distributed, the public key. If data is encrypted with one key, it must be decrypted with the other. For example, Alice wishes to send a private message, $m$, to Bob. She is in possession of Bob’s public key, $\text{pk}_B$. She encrypts the message using this key to obtain the ciphertext, $c = \text{Enc}(\text{pk}_B, m)$. To decrypt the message, the decryption function, Dec, require Bob’s private key, $\text{sk}_B$, as input to retrieve the plaintext, i.e. $m = \text{Dec}(\text{sk}_B, c)$. Using the public key to decrypt the message will render meaningless output. Now suppose Bob wishes to send a public message, $m'$, to Alice but he wants Alice to be sure that
he authored it. Therefore, he encrypts $m'$ with this private key, $sk_B$, to obtain the ciphertext, $m' = \text{Enc}(sk_B, m')$. As the scheme is asymmetric, in order to decrypt the message, Bob’s public key must be used. Alice has Bob’s public key and decrypts the ciphertext to retrieve $m' = \text{Dec}(pk_B, c')$. It should be noted, that as Bob’s public key is freely available, anyone can decrypt $c'$. However, as Bob is the only individual who has the private key, he must have encrypted the message. Therefore, the ciphertext is a digital signature of the message, proving authorship. We call this latter application of public key encryption signing.

Asymmetric cryptographic methods commonly use one-way functions, in particular trapdoor one-way functions. A one-way function is easy to compute in one direction. However, the inverse is believed to be hard to compute. “Easy” and “hard” are understood in the context of computational complexity theory, particularly polynomial time problems. A trapdoor function is a function whose inverse is believed to be hard to compute without knowledge of some secret information, the “trapdoor”, yet easy to compute with the “trapdoor”. Such a function makes it possible to safely disclose the method used to encipher the plaintext to a recipient. Even if an attacker obtains the ciphertext and the encryption function, he will be unable to compute the inverse function. Public key encryption schemes often exploit number theory, in particular, the properties of finite fields and elliptic curves over finite fields to derive a trapdoor function. For finite field methods, such as Diffie-Hellman [110] and ElGamal [127], it is believed that calculating a discrete logarithm of a number in a finite field is hard. The encryption scheme exploits this by having the private key, $x$, be the exponent used in an exponentiation process to compute the public key, $y$, i.e. $y = \alpha^x$, where $\alpha$ is an appropriate number chosen from the finite field. An encryption function is constructed which requires knowledge of the private key to invert. This satisfies the properties of a trapdoor function as an attacker would have to compute the discrete logarithm of $y$ in order to retrieve $x$ from $y$, and then decrypt the message. Elliptic curve cryptography [193, 233] exploits a similar property of elliptic curves over finite fields. In this case, given two points $P$ and $Q$ on an elliptic curve and an “addition” operation for points on the curve so that $P + Q$ is also a point on the curve, then computing $n$ such that $nP = Q$ is believed to be hard. Using this fact, a trapdoor function can be constructed. Like Diffie-Hellman and ElGamal, Rivest-Shamir-Adleman (RSA) [285] also uses exponentiation in a finite field for encryption and decryption. However, the property exploited to create the trapdoor function is
3.1. INTRODUCTION

that factorisation of a large number into its prime factors is hard.

Another form of one-way function used in cryptography is a cryptographic hash function \([274]\), such as MD5 message digest algorithm \([286]\) or Secure Hash Algorithm (SHA) \([240]\). A cryptographic hash function is a stronger form of hash function. Hash functions accept variable length inputs and output fixed size values. A hash function should have the property that if the input data is changed then this results, with high probability, in a change in the hash code. Cryptographic hash functions add further requirements. For a cryptographic hash function, it should be computationally infeasible to find an input that maps to a specified hash result (the one-way property) or to find two inputs that map to the same hash result (the collision-free property). As a cryptographic hash function is strictly one-way, a hash value cannot be decrypted to retrieve the plaintext. It should be noted that the hash value bit size is usually smaller than the input bit size. This means that a cryptographic hash function is, typically, a many-to-one mapping. If \( h = H(x) \), where \( H \) is the cryptographic hash function and \( x \) is the input, then a collision occurs if we can find \( y \neq x \) s.t \( H(x) = H(y) \). The collision-free property of the function ensures that finding such a pair, \((x, y)\), is computationally infeasible. Using a brute force method, the level of effort expended to search for a value \( x \) such that \( h = H(x) \) for a given hash value \( h \) of bit length \( M \) is proportional to \( 2^M \). One would assume that the level of effort to find a collision would be the same. However, the birthday paradox \([49, 359]\) shows that, with probability greater than 0.5, a collision will be found after \( 2^{M/2} \) attempts. Hash functions, both cryptographic and otherwise, are typically used to ensure data integrity. If a hash value for a given input changes from a previously recorded value, we can, with high probability, expect that the input has been modified. Therefore we can store a hash value and use it to detect modifications to the input data.

A message authentication code (MAC) is similar to a cryptographic hash \([178]\). Like a hash function, it is a many-to-one function that outputs a fixed length tag, \( t \). However, whereas a cryptographic hash function has one input, a MAC function, \( \text{MAC} \), takes two inputs, the variable length input, \( m \), and a secret key, \( k \), i.e. \( t = \text{MAC}(k, m) \). In order for a recipient to verify a MAC tag, the secret key, \( k \), must be shared between sender and receiver. The requirements for a MAC algorithm are similar to those of a cryptographic hash algorithm with the exception
that a MAC need not be a one-way function. The first requirement is that given an input \( m \) and \( \text{MAC}(k, m) \) it is computationally infeasible to find \( m' \) such that \( \text{MAC}(k, m) = \text{MAC}(k, m') \). This is similar to the collision-free property of cryptographic hash functions. The second requirement is that for \( m, m' \), randomly chosen from the domain of MAC, the probability that \( \text{MAC}(k, m) = \text{MAC}(k, m') \) is \( 2^{-n} \) where \( n \) is the bit size of the tag. The third requirement, is that, for \( m \) and \( m' \), where \( m' = f(m) \), the probability that \( \text{MAC}(k, m) = \text{MAC}(k, m') \) is also \( 2^{-n} \). These last two requirements are similar to the properties of general uniform hash functions. The final requirement is that given one or more pairs \((x_i, \text{MAC}(k, x_i))\) it is infeasible to compute a pair \((x, \text{MAC}(k, x))\) where \( x \neq x_i \). This requirement specifies that an attacker that has samples of plaintext and their associated MACs should not be able to compute a MAC for an arbitrary input. Again, like hash functions and cryptographic hash functions, MACs can be used to verify the integrity of data.

### 3.2 Applicable Cryptographic Methods

In this section we discuss cryptographic methods which are particularly applicable to MapReduce.

#### 3.2.1 Homomorphic Encryption

A homomorphic cryptographic system is one where an operation performed on ciphertexts decrypt to a result which is the same as the equivalent operation performed on the plaintexts, i.e.

\[
\text{Dec}(o(\text{Enc}(m), \text{Enc}(m'))) = o'(m, m')
\]

Where \( \text{Enc} \) is the encryption function, \( \text{Dec} \) is the decryption function, \( o \) is an operation on ciphertext, \( o' \) is the equivalent operation on plaintext, and \( m, m' \) are plaintexts.

Such a system would allow computation to be performed on encrypted data, providing the cryptographic system was homomorphic over the type of operations we wished to perform. We can see that this would be of immediate applicability
3.2. APPLICABLE CRYPTOGRAPHIC METHODS

to computation in the cloud where we would like to perform computation on sensitive data without exposing that data to a malicious third party.

In 1978, Rivest et al. \cite{287} proposed a set of requirements for such a homomorphic system, which they call a privacy homomorphism. To paraphrase \cite{287}, suppose we have an algebraic system \( M = (S, F, P, L) \), where \( S \) is a set, \( F \) is a set of operations \( \{f_1, \ldots, f_k\} \) on \( S \), \( P \) is a set of predicates \( \{p_1, \ldots, p_l\} \) on \( S \), and \( L \) is a set of constants in \( S \), \( \{l_1, \ldots, l_m\} \). Now, we also have the system \( C = (S', F', P', L') \) where each element of \( S \) is mapped to an element of \( S' \), each element of \( F \) to an element of \( F' \), and so on. We denote the mapping from \( M \) to \( C \) as encryption and \( C \) to \( M \) as decryption. If \( \phi : S' \to S \), the decryption of elements of \( S' \) to elements of \( S \), exists then so should \( \phi^{-1} : S \to S' \), the encryption function. For \( \phi \) to be a homomorphism the following must hold true:

1. \( \forall i : \phi(s'_i) = s_i \)
2. \( \forall i, s'_1, s'_2, \ldots : \phi(f'_i(s'_1, s'_2, \ldots)) = f_i(\phi(s'_1), \phi(s'_2), \ldots) = f_i(s_1, s_2, \ldots) \)
3. \( \forall i, s'_1, s'_2, \ldots : \phi(p'_i(s'_1, s'_2, \ldots)) \equiv p_i(s_1, s_2 \ldots) \)

To satisfy Rivest et al.’s requirements for a privacy homomorphism the following must hold true:

1. \( \phi \) and \( \phi^{-1} \) should be easy to compute.
2. The operations \( f'_i \) and \( p'_i \) should be efficiently computable.
3. The encryption of \( s_i, \phi^{-1}(s_i) \), should not require much more space to represent than \( s_i \).
4. Knowledge of \( \phi^{-1}(s_i) \) for many \( s_i \) should not reveal \( \phi \) (a ciphertext-only attack), i.e. statistical methods cannot be used to deduce \( \phi \) from a set of ciphertexts.
5. Knowledge of \( s_i \) and \( \phi^{-1}(s_i) \) for several values of \( s_i \) should not reveal \( \phi \) (a chosen plaintext attack).
6. The operations and predicates in \( C \) should not be sufficient to efficiently compute \( \phi \).
**CHAPTER 3. SURVEY**

**Partially Homomorphic Encryption (PHE)**

There are several encryption schemes that demonstrate homomorphic properties for a limited set of operations. We denote these as *partially homomorphic*.

Unpadded RSA \[285\] and ElGamal \[127\] are multiplicatively homomorphic. The Goldwasser-Micali system \[146\] is homomorphic over addition modulo 2 (the XOR operation). Cryptosystems related to Goldwasser-Micali, such as Benaloh \[38\], Naccache-Stern \[241\], Okamoto-Uchiyama \[253\], and Paillier \[262\] are also additively homomorphic. Ishai-Paskin \[172\] allows for polynomial size branching programs, such as finite state automata or circuits, to be computed on bits encrypted by an additively homomorphic scheme. A comprehensive survey of partially homomorphic encryption scheme is given in \[1\].

**Somewhat Homomorphic Encryption (SHE)**

We can consider somewhat homomorphic encryption system (SHE) as an intermediate stage between PHE and fully homomorphic encryption. A SWHE is a cryptosystem which is homomorphic for a range of operations but only for a limited set of inputs or functions that may be computed. In particular, if we think of computation as a Boolean or arithmetic circuit \[23\], then a SHE scheme can only compute circuits of a certain depth before the ciphertext cannot be successfully decrypted.

As many SHE schemes are the building block toward a fully homomorphic system, important contributions are discussed in the section below.

**Fully Homomorphic Encryption (FHE)**

Whether a fully homomorphic cryptographic system could be realised remained an open question in cryptography until 2009. In \[137\], Craig Gentry devised a fully homomorphic scheme using ideal lattices.

Gentry’s scheme starts off by devising a SHE scheme where the ciphertext consists of the plaintext modified by a “noise” parameter. This “noise” grows as ciphertexts are added and multiplied until there comes a point where the plaintext cannot be recovered from the ciphertext. Gentry then represents a computation on
the ciphertext as an arithmetised Boolean circuit of additions and multiplications \cite{23}. As a result of the “noise” we use to encipher the plaintext, this arithmetic circuit must be limited in its depth to prevent the “noise” parameter growing too large. Therefore, this allows for low degree polynomials to be computed where the degree of the polynomial is the depth of the circuit.

This “somewhat” homomorphic scheme is then “bootstrapped” to a fully homomorphic scheme by the observation that the decryption algorithm for the ciphertext may be represented as an arithmetic circuit. Therefore, we are able to periodically “refresh” the ciphertext by decrypting and re-encrypting before the noise parameter grows too large. Now, to prevent the plaintext being exposed, we can make this “refreshing” process recursive, so that at each “refresh” point, the decryption and encryption is performed homomorphically, allowing us to compute circuits of arbitrary depth.

Furthermore, Gentry shows that circuits deeper that the bootstrapping process can deal with may be transformed to a shallower circuit (“squashing”), thereby allowing the bootstrapping process to be performed. As a trade-off, “squashing” increases the size of the ciphertext.

Gentry’s initial work was performed using ideal lattices as this led to decryption circuits of low complexity (such as an inner product or matrix-vector multiplication). This allows for the bootstrapping process to be performed. RSA, El-Gamal, Paillier, and other similar homomorphic systems use exponentiation or other techniques, which lead to complex decryption circuits. Security of Gentry’s scheme is based on the assumed hardness of two problems: the sparse subset sum problem and a worst-case problem on lattices. However, Gentry’s scheme seems to be impractical as increasing the security parameter leads to rapid blow-up in the size of the ciphertext and computation time. Optimisations of this scheme were developed by Stehlé and Steinfeld \cite{310}, Smart and Vercauteren \cite{304,305}, and Gentry and Halevi \cite{138,139}. The latter provided an implementation of the Gentry scheme. However, for large dimensions of the lattice, and, therefore, greater security, this implementation required 30 minutes to perform one bootstrapping operation and had a public key size of 2.3 GB.

Since Gentry’s initial breakthrough there have been several developments of this lattice-based fully homomorphic scheme. Starting with work by Brakerski,
Gentry, and Vaikuntanathan \cite{65,66} there have been several significant improvements which use the hardness of the “learning with errors” (LWE) problem \cite{281} to provide security. Brakerski, Gentry and Vaikuntanathan’s breakthrough was to apply a modulus switching technique after every multiplication. In this case, suppose we start with a modulo $p_1$ ciphertext with noise $2^\rho$. After one multiplication the noise grows to $2^{2\rho}$. Now, if we switch to a new modulus $p_2$, where $p_1/p_2 \approx 2^{-\rho}$, then we produce a modulo $p_2$ ciphertext with noise $2^\rho$. Therefore, using a ladder of moduli, one can perform greater levels of multiplication before the noise grows large, as the greatest modulus required grows linearly with the number of multiplications. In \cite{64}, Brakerski devised a scale-invariant FHE system that did not require modulus switching based on the hardness of the GapSVP problem \cite{267}. Fan and Vercauteren ported Brakerski’s work to a system based on the ring LWE problem. Gentry et al. \cite{143} devised a simpler attribute-based system where the evaluation key is dispensed with in favour of an identity-based scheme. In \cite{143}, the target identity can evaluate a circuit using the public parameters. Brakerski and Vaikuntanathan \cite{68} then devised a scheme based on the polynomial LWE problem. All of these more recent schemes are considerably more efficient than Gentry’s initial scheme, the ciphertext “noise” grows more slowly allowing for greater depth circuits to be evaluated before bootstrapping is required. Indeed, Brakerski and Vaikuntanathan \cite{68} showed that for certain circuits, the noise grows even more slowly for the Gentry-Sahai-Waters scheme \cite{143}. Optimisations by Gentry, Halevi, and Smart \cite{140,142} and Alperin-Sheriff and Peikert \cite{7} have reduced the complexity of evaluating circuits on data encrypted by these methods to an overhead that is polylogarithmic in the security parameter $\lambda$. Recently, Cheon et al. \cite{83} devised a scheme based on their extension of the approximate common divisor problem \cite{164}, polynomial approximate common divisors, that is capable of computing polynomials on integers efficiently. Kim et al. \cite{190} have extended Cheon et al.’s work to only reveal partial information about the underlying message space, which they call message space hidability.

Other non-lattice based systems have been devised. van Dijk et al. \cite{111} produced a fully homomorphic scheme over the integers where a simple “somewhat” homomorphic encryption scheme is bootstrapped to a fully homomorphic scheme. Coron et al. \cite{81,95,97} produced several refinements of this scheme. Lopez-Alt, Tromer, and Vaikuntanathan \cite{213} devised an alternative efficient scheme that is based on NTRU \cite{163}. Acar et al. \cite{1} detail other work derived from van Dijk et
3.2. APPLICABLE CRYPTOGRAPHIC METHODS

al. and Lopez-Alt et al.

Despite these significant advancements, computation on fully homomorphically
encrypted data is currently highly time and space inefficient compared to compu-
tation on unencrypted data. The HELib \cite{157} implementation of the Brakerski-
Gentry-Vaikuntanathan scheme \cite{65} with the Gentry-Halevi-Smart optimizations
\cite{141} performs a bootstrapping operation in 5-10 minutes on a “packed cipher-
text” that encrypts a vector of 1024 plaintext values \cite{156}. The FHEW \cite{117}
implementation of the scheme described in \cite{116} reports bootstrapping a single
NAND operation in under a second \cite{116}. However, this requires 2.2 GB of
memory to perform. Using the bootstrapping technique of FHEW, Chillotti et
al. \cite{85, 86} reduce the bootstrapping time to less than 0.1s for the Gentry-Sahai-
Waters (GSW) scheme \cite{143}. They achieve this by replacing the internal GSW
ciphertext product with an external product of a GSW ciphertext and a cipher-
text of a scale invariant ring LWE scheme such as \cite{80}.

Again, \cite{1} provides a comprehensive survey of SHE and FHE schemes.

3.2.2 Secret Sharing

A major problem of using cryptographic techniques in a distributed environment
is the secure storage and distribution of cryptographic keys. In particular, if we
wish to share a secret key among many nodes in a distributed environment, how
might this be achieved so that knowledge of a share on one node, does not expose
the secret?

Independently invented in 1979 by Shamir \cite{297} and Blakley \cite{45}, secret sharing
describes a system where a master secret is divided into \( n \) parts by the dealer,
called shares. One share is distributed to each participant (player). To recover
the master key, knowledge of \( t \) shares is required. In Shamir’s solution, the master
secret is \( p(0) \), where \( p \) is a polynomial of degree \( t - 1 \). The shares are \( p(i) \) where
\( i \in \{1, \ldots, n\} \). Recovery of the master secret is achieved by interpolating using \( t \)
shares.

Blakley’s scheme uses hyperplanes. In this version, the master secret is derived
as one of the co-ordinates of the point of intersection of \( t \) \((t - 1)\)-dimensional
hyperplanes. Each of the \( n \) shares is the information to describe a hyperplane.
The secret is recovered by knowledge of $t$ hyperplanes, calculating their point of intersection, and then selecting the appropriate co-ordinate. Blakley’s scheme is considerably more space inefficient than Shamir’s, each share is $t$ times larger than the secret.

The Chinese Remainder Theorem has also been used as a means of secret sharing \cite{21, 231}. In these schemes, given $n$ relatively prime numbers $m_1, \ldots, m_n$, we choose the secret, $S$, such that $S$ is smaller than the product of any $t$ of these numbers. The shares are residues of $S$ modulo $m_i$, where $i \in \{1, \ldots, n\}$. We recover $S$, using $t$ residues, by the Chinese Remainder Theorem.

There are a couple of problems with secret sharing schemes as initially presented. Firstly, what if we have cheating players who do not correctly disclose their share? To counter this, verifiable secret sharing was developed \cite{87, 130, 277}. Such a scheme allows participants to verify that others are not lying about their shares.

Secondly, if an attacker can break into $t$ nodes and retrieve the shares, then they will be able to compute the secret. Proactive secret sharing \cite{161} was developed to counter this threat. In this variant of the Shamir scheme, the shares are updated. Each player $i$, periodically computes a random polynomial $\delta_i(x)$, and secretly sends $\delta_{i,j} = \delta_i(j)$ to each other participant. Each player computes its new share value as $x_i^{(t)} = x_i^{(t-1)} + \sum_j \delta_{j,i}$, where $t$ is the time period since initialisation. In this scheme, an attacker must obtain $k$ non-updated shares at a particular point in time in order to compute the secret.

A comprehensive survey of secret sharing schemes is given in \cite{27}.

### 3.2.3 Order-Preserving Encryption

As the name suggests, order-preserving encryption (OPE) preserves the ordering of data in the plaintext space into the ciphertext space. Order-preserving encryption was devised to allow the encrypted data to be sorted so that the ordering produced was meaningful once decrypted. This allows sorting to occur without revealing the entirety of the underlying plaintexts.

A function, $f : A \rightarrow B$, where $|A| \leq |B|$ is said to be order-preserving (or monotonically increasing) if $x_1 > x_2$, $x_1, x_2 \in A$, then $f(x_1) > f(x_2)$. Therefore
3.2. APPLICABLE CRYPTOGRAPHIC METHODS

an order-preserving encryption system is defined as \( \{K, \text{Enc}, \text{Dec}\} \) where \( K \) is a set of secret keys, and \( \text{Enc}, \text{Dec} \) are order-preserving functions such that \( \text{Enc} : K \times M \rightarrow C \) and \( \text{Dec} : K \times C \rightarrow M \), where \( M \) is the plaintext space and \( C \) is the ciphertext space.

Prior to Boldyreva et al. [53], OPE had been investigated by Agrawal et al. [4] and others (see [4] for earlier references). However, it wasn’t until Boldyreva et al. that it was claimed that an OPE scheme was provably secure. Boldyreva et al.’s algorithm constructs a random order-preserving function by mapping \( M \) consecutive integers in a domain to integers in a much larger range \([1, N]\), by recursively dividing the range into \( M \) monotonically increasing subranges. Each integer is assigned a pseudorandom value in its subrange. The algorithm recursively bisects the range, at each recursion sampling from the domain until it hits the input plaintext value. The algorithm is designed this way because Boldyreva et al. wish to sample uniformly from the range. This would require sampling from the negative hypergeometric distribution, for which no efficient exact algorithm is known. Therefore they sample the domain from the hypergeometric instead. As a result, each encryption requires at least \( \log N \) recursions. Furthermore, so that a value can be decrypted, the pseudorandom values generated must be reconstructible. Therefore, for each instance of the algorithm, a plaintext will always encrypt to the same ciphertext. This implies that the encryption of low entropy data might be very easy to break by a “sorting” attack [242]. In [53], the authors claim that \( N = 2M \), a claim repeated in [79], although [52] suggests \( N \geq 7M \). However both [52, 53] take no account of \( n \), the number of values to be encrypted. The scheme should have \( n \ll M \) to avoid the sorting attack of [242]. If \( c = f(m) \) is Boldyreva et al.’s OPE, it is straightforward to show that we can estimate \( f^{-1}(c) \) by \( \hat{m} = Mc/N \), with standard deviation approximately \( \sqrt{2\hat{m}(1 - \hat{m}/M)} \). For this reason, Boldyreva et al.’s scheme always leaks about half the plaintext bits.

Yum et al. [358] extend Boldyreva et al.’s work to non-uniformly distributed plaintexts. This can improve the situation in the event that the client knows the distribution of plaintexts. This “flattening” idea already appears in [4]. In 8.2.3 we discuss a similar idea.

In [52], Boldyreva et al. suggest an extension to their original scheme, modular order-preserving encryption (MOPE), by simply transforming the plaintext before
encryption by adding a term modulo $M$. The idea is to cope with some of the problems discussed above, but any additional security arises only from this term being unknown. Note also that this construction again always produces the same ciphertext value for each plaintext.

Teranishi et al.\[312\] devise a new OPE scheme that satisfies their own security model. However, their algorithms are less efficient, being linear in the size of the message space. Furthermore, like Boldyreva et al., a plaintext always encrypts to the same ciphertext value.

Krendelev et al.\[196\] devise an OPE scheme based on a coding of an integer as the real number $\sum b_i 2^{-i}$ where $b_i$ is the $i$th bit of the integer. The algorithm to encode the integer is $O(n)$ where $n$ is the number of bits in the integer. Using this encoding, they construct a matrix-based OPE scheme where a plaintext is encrypted as a tuple $(r, k, t)$. Each element of the tuple is the sum of elements from a matrix derived from the private key matrices $\sigma$ and $A$. Their algorithms are especially expensive, as they require computation of powers of the matrix $A$. Furthermore, each plaintext value always encrypts to the same ciphertext value.

Khadem et al.\[179\] propose a scheme to encrypt equal plaintext values to differing values. Their scheme is similar to Boldyreva et al. where a plaintext is mapped to a pseudorandom value in a subrange. However, this scheme relies on the domain being a set of consecutive integers for decryption.

Liu et al.\[212\] addresses frequency of plaintext values by mapping the plaintext value to a value in an extended message space and splitting the message and ciphertext spaces nonlinearly. As in our scheme, decryption is a simple division. However, the ciphertext interval must first be located for a given ciphertext which is $\Omega(\log n)$ when $n$ is the total number of intervals.

Liu and Wang\[211\] describe a system where random “noise” is added to a linear transformation of the plaintext. However, in their examples, the parameters and noise used are real numbers. The security of such a scheme is unclear.

In\[271\], Popa et al. discuss an interactive protocol for constructing a binary index of ciphertexts. Although this protocol guarantees ideal security, in that it only reveals the ordering, it is not an OPE. The ciphertexts do not preserve the ordering of the plaintexts, rather the protocol requires a secure client to decrypt the ciphertexts, compare the plaintexts, and return the ordering. It is essentially
equivalent to sorting the plaintexts on the secure client and then encrypting them. Popa et al.’s protocol has a high communication cost: $\Omega(n \log n)$. This may be suitable for a database server where the comparisons may be made in a secure processing unit with fast bus communication. However, it is unsuitable for a large scale distributed system where the cost of communication will become prohibitive. Kerschbaum and Schroepfer \cite{189} improved the communication cost of Popa et al.’s protocol to $\Omega(n)$ under the assumption that the input is random. However, this is still onerous for distributed systems. Kerschbaum \cite{188} further extends this protocol to hide the frequency of plaintexts. Boelter et al.\cite{50} extend Popa et al.’s idea by using “garbled circuits” to obfuscate comparisons. However, the circuits can only be used once, so their system is one-time use.

**Order-Revealing Encryption (ORE)**

Also of note is order-revealing encryption (ORE), a generalisation of OPE introduced by Boneh et al. \cite{56}, that only reveals the order of pairs of ciphertexts. An ORE is a scheme $(C, E, D)$ where $C$ is a comparator function that takes two ciphertext inputs and outputs ‘$<$’ or ‘$\geq$’, and $E$ and $D$ are encryption and decryption functions. This attempts to replace the secure client’s responsibility for plaintext comparisons in Popa’s scheme with an exposed function acting on the ciphertexts.

Boneh et al.’s construction uses multilinear maps. However, as stated in Chenette et al. \cite{79}, “The main drawback of the Boneh et al. ORE construction is that it relies on complicated tools and strong assumptions on these tools, and as such, is currently impractical to implement”.

Chenette et al. offer a more practical construction, with weaker claims to provable security. However, since it encrypts the plaintexts bit-wise, it requires a number of applications of a pseudorandom function $f$ linear in the bit size of the plaintext to encrypt an integer. The security and efficiency of this scheme depends on which pseudorandom function $f$ is chosen.

Lewi and Wu \cite{202} devise an ORE scheme where there are two modes of encryption: left and right. The left encryption consists of a permutation of the domain and a key generated by hashing the permuted plaintext value. The right ciphertext consists of encryptions of the comparison with every other value in the
domain. It is a tuple of size $d + 1$ where $d$ is the size of the domain. Lewi et al. then extend this scheme to domains of size $d^n$. This results in right ciphertext tuples of size $dn + 1$.

The security of these ORE schemes is proven under a scenario similar to IND-OCPA [53] (see section 8.2.2). However, under realistic assumptions on what an adversary might do, these ORE schemes seem to have little security advantage over OPE schemes. For example, in $O(n \log n)$ comparisons an adversary can obtain a total ordering of the ciphertexts, and, hence the total ordering of the plaintexts. A disadvantage of ORE schemes are that they permit an equality test on ciphertexts [56, p.2] by using two comparisons. This could be used to aid a guessing attack on low-entropy plaintexts, e.g. [118, 242]. On the other hand, the information leakage of the ORE schemes so far proposed appears to be near-optimal.

### 3.2.4 Searchable Symmetric Encryption

Searchable symmetric encryption (SSE) is a symmetric encryption system that allows keyword searches to be performed over the generated ciphertexts. The concept was introduced by the Song et al. paper in 2000 [309]. Their scheme first encrypts data using a deterministic encryption system. A pseudorandom bit stream of length $l - t$ is generated where $l$ is the bit length of the deterministic ciphertext and $t$ is the bit length of a tag generated by a MAC algorithm. A MAC of the bit stream is generated to produce a tag which is appended to the end of the stream. This is then XORed with the deterministic ciphertext to produce the final ciphertext. The addition of the pseudorandom data masks the underlying deterministic ciphertext and ensures that two ciphertexts of the same plaintext are different. To search over ciphertext for a keyword, the deterministic ciphertext of the keyword and the MAC key are supplied. The ciphertext is XORed with the deterministic ciphertext. The resulting value is divided into sections of $l$ and $t$ bits. A MAC of the first $l$ bits is produced and compared with the last $t$ bits. If they are identical, then it is a successful match. Song et al. show that their system is a secure pseudorandom generator. Indeed, in [184], it is shown that Song et al.’s scheme is IND-CPA secure. A limitation of Song et al.’s scheme is that the reliance on a pseudorandom stream to mask the plaintext requires that
words are delivered in the same order for decryption that they were encrypted in. Amanatidis et al. [8] address this by using the “encrypt-and-mac” paradigm. The keyword is encrypted using a pseudorandom cipher that produces different ciphertext for encryptions of the same plaintext. A tag generated by a MAC algorithm is also produced. The tag, of length $t$, is appended to the ciphertext produced by the cipher. To search, a tag is generated for the search keyword. If the last $t$ bits of the ciphertext are the same as the search tag, then it denotes a successful match. However, despite being a stateless system, the MAC algorithm is deterministic, which makes the system only as strong as a deterministic cipher.

Searching in Song et al.’s scheme is linear in the number of keywords in a document. To assist faster searching over large collections of documents, Goh devised the idea of a secure index in 2003 [144]. Goh uses a Bloom filter [48] as an index for each document. This makes the search time linear in the number of documents. However, the time to build the indices is linear in the number of words in each document. Goh also devises the first security definitions for SSE.

Bellovin and Cheswick [36] extend Goh’s idea to a multi-user scenario using a group cipher [270]. Chang and Mitzenmacher [75] expand on Goh’s work by using a pre-processed dictionary of keywords. Again, indices are built per document. Each index is a $m$-bit array ($m$ is the number of keywords) with a one indicating the presence of the keyword in the document.

Curtmola et al. [100] produce the first scheme that has optimal (sublinear) search time. Their idea is to use an inverted index for each keyword in the dictionary rather than for each document. Each keyword index lists the documents in which the keyword appears. Van Liesdonk et al. [206] extends this idea to allow for the index to be efficiently updated. Kamara et al. [184] further extend Curtmola’s idea to allow for dynamic updates.

Kamara and Papamanthou [183] devise a new dynamic scheme that is highly parallelizable. They use a structure similar to a binary tree where each leaf is a pointer to a file as the basis of the per-keyword index.

Golle et al. [150] devised the first scheme that supported conjunctive searches. A document is assumed to have several fields which are used for keywords (such as header fields). Each document is encrypted as a vector of words. Hence, searching is linear in the number of words in all documents. Others have built on this work [25, 69, 182, 234, 290, 335], including Cash et al. [72] who produced the first
scheme with sublinear search time using an inverted index approach. Fuzzy and similarity searches have also proved a fertile area of investigation with several papers published, including [2, 61, 204, 264, 299].

Comprehensive surveys of secure searchable encryption are provided in [60, 181, 342].

3.3 Related Works

In this section, we examine related works in the fields of cryptographic approaches to MapReduce security, MapReduce result verification, and more general approaches to the problem.

3.3.1 Computing on Encrypted Data

There is a body of work relating to queries on encrypted databases [55, 152, 153, 306, 307, 319, 343] but relatively little concerning MapReduce computation on encrypted data specifically.

In this section we concentrate on three, related, practical approaches to computing on encrypted data, two of which relate directly to MapReduce. The first, CryptDB [272], is the direct inspiration of MrCrypt [313] and Crypsis [311].

CryptDB

CryptDB allows for queries to be processed on an encrypted SQL database. CryptDB takes a pragmatic approach to the problem by choosing an appropriate encryption scheme for each data field and the operation to be performed on it.

Popa et al. [272] use the following encryption schemes:

**Random (RND)** The strongest form of encryption, which is indistinguishable when subjected to an adaptive chosen plaintext attack (IND-CPA) [35]. This is implemented using AES [237] or Blowfish [294] in CBC mode [125].

**Deterministic (DET)** A weaker form of encryption to allow for equality to be
3.3. RELATED WORKS

determined over encrypted data. This is implemented as AES in CMC mode with a constant (zero) initialisation vector.

Order-preserving (OPE) The scheme from is implemented.

Homomorphic (HOM) Popa et al. implement the Paillier scheme to support addition operations. They use a 2048-bit key size. They do not implement a method to support multiplication.

Join (JOIN and OPE-JOIN) Popa et al. present a novel scheme to support joins which is described below.

Text Search (SEARCH) Popa et al. implement Song et al. CryptDB adjusts the encryption of a field dynamically to support the requested query. For example, a the SQL query `SELECT ID FROM Employees WHERE Name = Alice` requires that the Name is encrypted using the deterministic scheme. CryptDB converts the column to the required encryption type.

The cryptographic primitive to support joins is constructed using elliptic curve cryptography. The columns are initially encrypted by the function JOIN-ADJ as follows:

\[ \text{JOIN-ADJ}(K, m) = P^{K \cdot PRF_{K_0}(m)} \]

where \( K \) is a secret key, \( m \) is the plaintext value, \( P \) is a point on an elliptic curve, \( PRF \) is a pseudorandom function, and \( K_0 \) is a key derived from a master key. \( K_0 \) is the same across all columns of the database table. The exponentiation operation is shorthand for the elliptic curve group addition operation.

When a join is requested between two columns \( c, c' \) each encrypted using keys \( K \) and \( K' \) respectively, a trusted third-party (the client or a proxy) computes \( \Delta K = K \cdot K'^{-1} \). \( c' \) is then transformed by the following procedure:

\[
\text{JOIN-ADJ}(K', m')^{\Delta K} = (P^{K' \cdot PRF_{K_0}(m')})^{K \cdot K'^{-1}} = P^{K \cdot PRF_{K_0}(m')} = \text{JOIN-ADJ}(K, m')
\]

Now \( c \) and \( c' \) are encrypted using the same key and an equijoin can be performed.
The trade-off for CryptDB’s pragmatic approach is a weakening of security. Not only may we have to encrypt data using weaker forms of encryption, the type of encryption leaks information about the data encrypted. A field encrypted using DET informs a third-party that it is equality comparable, ruling out that is, say, a BLOB field.

CryptDB tries to mitigate against decrypting and re-encrypting as it performs “on the fly” conversion of a field from one encryption scheme to another by encrypting fields in an “onion skin” where each field is encrypted several times using the various schemes. However, as fields are updated, eventually a field will have to be decrypted in order to convert it to another encryption type. At this point, the data will be exposed.

Finally, the encryption schemes employed for HOM and JOIN are computationally expensive. In both cases, an entire column is encrypted by a relatively expensive method. For large databases, the impact on performance will be considerable.

MrCrypt

MrCrypt takes inspiration from CryptDB. However, MrCrypt transforms the user-defined Map and Reduce functions so that they can operate on encrypted data. MrCrypt examines the source code and infers the encryption scheme for each variable and constant. It then outputs a modified program where data and operations are replaced by their counterparts for encrypted data.

MrCrypt employs many of the same cryptographic schemes as Popa et al. To support multiplication, Crypsis adds a multiplicatively homomorphic scheme (MHE) using ElGamal. In their hierarchy of encryption schemes, it is noted that to perform addition and multiplication in the same operation would require a fully homomorphic scheme, which they do not implement.
3.3. RELATED WORKS

Crypsis

In essence, Crypsis \cite{311} implements CryptDB for the MapReduce paradigm. Crypsis exploits Apache Pig \cite{18}, a runtime for transforming scripts, in the SQL-like Pig Latin language, into user-defined functions (UDFs) which may be deployed on an unmodified Hadoop \cite{17} system. Crypsis uses the many of the same encryption schemes as CryptDB and MrCrypt. However, it drops Popa et al.’s JOIN in favour of the DET scheme.

Crypsis modifies the Pig runtime so that it produces UDFs which can operation on encrypted data. Like Pig, these UDFs can be employed in an unmodified Hadoop environment. As with CryptDB, when an operation requires the encryption scheme of a field to be changed, Crypsis does this dynamically. Crypsis ships the data to a trusted third party for re-encryption. However, this proves to be a flaw in Crypsis’ design because in a large distributed environment, such as MapReduce, transfer and re-encryption of data contributes significantly to latency. In the case of computing an inner product, Crypsis’ design would be prohibitively slow as it would have to first compute the products homomorphically using the ElGamal scheme, then ship them to the client to be re-encrypted using the Paillier scheme, then perform the additions homomorphically.

Other MapReduce Based Approaches

Zhu and Bao \cite{367} detail their implementation of a private information retrieval scheme for MapReduce. This scheme filters the intermediate data produced by the mappers before passing it to the reducers. They use a Bloom-Filter with Storage \cite{260} combined with the Paillier additively homomorphic encryption scheme \cite{262} to implement the privacy preserving filter.

In \cite{326}, Varadharajan and Kuppusamy discuss their modified MapReduce framework to support searchable encryption. In their system, a secure agent generates a master key from this user keys are generated and distributed to the users. Data is first encrypted by users using a lightweight encryption scheme. This data is then transferred to the DFS where the distributed properties of MapReduce are leveraged to further encrypt the files. This second round of encryption uses the cluster wide master key. Searching is performed by the user first encrypting the
keyword using his user key. This partially encrypted ciphertext is passed to the secure agent which performs the second round of encryption. This encrypted keyword is then used to search across the encrypted files in the DFS. As they use modular arithmetic for encryption, it is not clear from their paper how the encrypted files might be searched using the encrypted keyword. Furthermore, it is also not clear how the two rounds of encryption allows user to search over other user’s files without sharing user keys as is claimed in the paper.

In [301], Shetty and Manjaiah describe a MapReduce-based method for computing an average of a numerical field, star ratings for movies, which has been encrypted. They use an modular arithmetic-based encryption scheme similar to our HE1N system (7.2.2) but where the ciphertext is $m + k \mod p$ where $k$ is the secret key, a large prime and $p$ is the product of $k$ and another large prime. Although not stated, clearly the plaintexts must be greater than zero. This encrypted data is uploaded to the cluster and further encrypted during the map phase by adding an additional noise term $rs$ where $r$ is a random integer and $s$ is a noise key shared by the mappers and reducers such that it is larger than any of the inputs. In the reduce phase, the noise is removed by computing $\mod s$. It is not clear how adding the noise term during the map phase enhances the security of the data since the data without the noise term is uploaded as input to the MapReduce cluster. Also, as they are computing an average value, presumably by summing all the ciphertexts to obtain a total which is decrypted and then divided by the number of inputs, it is not clear whether they have considered that this total might exceed $p$. Presumably, $p$ is chosen to be large enough that this will not happen.

### 3.3.2 Result Verification

In open systems such as clouds, service providers may come from different administrative domains that may not be trustworthy. In the specific case of MapReduce applications, workers may belong to several different administrative domains. This poses a risk that we may have malicious workers acting independently or in concert to compromise the integrity of the computation result.

In a wider context, several solutions to this problem have been proposed, such
as, interactive proofs\cite{148}, probabilistically checkable proofs\cite{20,24} and non-
interactive verifiable computing\cite{19,37,89,136}. However, as argued in\cite{296},
these techniques are currently impractical, leading to proofs which are too long
for a verifier to handle, too complex to be bug-free, or require an expression of the
program, which the verifier is able to check, such as a Boolean circuit, which is
inefficiently large. For example,\cite{136} uses Yao’s garbled circuits\cite{30} along with
fully homomorphic encryption as the expression of the program. For realistic com-
putations, such a representation would currently require infeasibly large storage
requirements and computing times. Therefore, in this section, we mainly examine
practical solutions for checking results of MapReduce computation. However, we
do detail\cite{84} which is based on zero-knowledge proofs.

Wei et al.’s\cite{346} approach to the aforementioned problem is their SecureMR
framework which implements a service integrity assurance mechanism for MapRe-
duce. It is a decentralised replication-based integrity verification scheme. Wei et
al. present an architecture and protocols to securely schedule and assign work to
worker nodes. Map workers authenticate the intermediate data partitions they
have generated by use of a digital signature. The master is notified of these
partitions and verifies the digital signatures using the map worker’s public key.
The map worker’s data is only designated as committed when the signature is
verified. Similarly, reduce workers use a verification protocol to ensure that the
data they have been allocated by the master is authentic. Protocol messages are
also authenticated by digital signatures. Authenticating messages before they are
processed helps prevent denial of service attacks on key participants, such as the
master node. Furthermore, the use of digital signatures provides non-repudiation
for protocol messages and intermediate data. However, their work assumes that
key pairs have been generated for all workers and that the public keys have been
distributed to the master node. This doesn’t address dynamic addition of cluster
nodes as is typical in cloud environments. They provide no mechanism to gener-
ate key pairs for new workers nor do they discuss how to securely distribute public
keys to the master or to assign a measure of trust to those keys. Their solution
demonstrates a performance penalty of 5% to 12%. However, their performance
tests are performed using Apache’s Hadoop on a very small cluster of 14 hosts
with approximately 100 concurrent programs running. They do not discuss how
the performance might scale to a more realistic application size of thousands of
concurrent programs.
Zhou et al. [366] approach this problem via an authenticated MapReduce application implemented on their DS2 platform. In their paper, each map worker provides an authenticator for the intermediate data it emits. A reduce worker will not accept intermediate data unless its authenticator can be verified. The security is provided by HMAC-SHA1 message authentication codes and RSA-1024 digital signatures and is compared with no security. On a cluster of 16 quad-core machines, their results exhibit a job completion overhead of 17% and 78% using HMAC-SHA1 and RSA-1024 respectively. Furthermore, their work does not address dynamic addition of cluster nodes nor does it address performance in large scale MapReduce clusters.

Xiao and Xiao [352] present a novel approach to the problem. Workers are divided into two classes: normal and speculative. Multiple speculative workers are assigned the same task. Speculative workers generate an MD5 hash for the intermediate data they generate. In their paper, speculative workers are forced to enter a committing phase before completion. The master node waits for all running workers assigned to a task to enter the committing phase. It compares the MD5 hashes received from all speculative workers for that task and chooses an MD5 hash for the correct result by majority decision. It then randomly chooses a speculative worker whose MD5 hash matches the one decided upon and kills all other workers. Although this is a lightweight approach to the problem it requires spawning multiple workers for the same task. This approach requires very large clusters where computation is replicated to guarantee output correctness. It also relies on the fact that the only possible source of unreliable data is the worker nodes themselves. It makes no provision to protect the messages sent between workers and master from alteration or fabrication. The performance analysis for speculating 10% and 30% of tasks shows an overhead of 41% and 72% respectively.

Xiao and Xiao [355] address the same problem using an accountability test for workers. Machines in a trusted domain form an auditor group which is responsible for checking the correctness of results produced by workers. Each map worker generates a tamper-proof log of task operations which may be replayed by an auditor to check correctness. Rather than duplicating all work from untrusted workers by replaying logs in entirety, Xiao and Xiao propose a probabilistic measure of accountability where a sample of records are reprocessed. They do not discuss the performance overhead required for performing these accountability tests.
3.3. RELATED WORKS

Wang et al. discuss their Verification-based Integrity Assurance Framework for MapReduce (VIAF). In this solution, each mapper must accumulate a reputation score before the master trusts it enough to release its results to the reducers. Each map task is assigned to two randomly chosen mappers. Each mapper processes the task, computes the results, computes a hash of the results, and stores it in a log they call the history cache. If the results from the two mappers disagree, the master will re-schedule the task on two other randomly chosen mappers. Otherwise, the master may choose to verify the result or to accept it. If verified or accepted, the two workers’ reputation scores are increased and the result is cached. When the two workers have achieved a sufficient reputation, then the result is released to the reducer. Their performance analysis show that VIAF doubles the execution time of the map phase, and that increasing the input size for TeraSort causes the execution time to grow polynomially. Wang et al. further adapt VIAF to hybrid clouds as Cross-cloud MapReduce (CCMR) and IntegrityMR. In CCMR, the workers are in the public cloud and the master and verifiers in a private cloud. In IntegrityMR, there are two sets of workers in different public clouds and the results from one set are verified against the other.

Moca et al. present a result verification method for MapReduce deployed in a desktop volunteer grid computing context. They use a majority voting method, where tasks are scheduled on multiple workers and the results generated are compared, in a similar fashion to.

Bendahmane et al. discuss a weighted majority voting mechanism where each worker result is assigned a weight based on its reputation score. Each map task is replicated to several workers and the results are collected by the reducer into groups of the same value. The reducer calculates a weighted score for each result group which is the sum of the reputation scores for each mapper that returned that particular result as:

$$w(G_j) = \frac{\sum_{i=1}^{r} w(P_i) : (v(P_i) = v(G_j))}{\sum_{i=1}^{r} w(P_i)}$$

where $w(G_j)$ is the weight of the $j$-th result group, $w(P_i)$ is the weighted reputation of worker $P_i$, $v(P_i)$ is the result that worker $P_i$ computed, $v(G_j)$ is the
common result of the $j$-th result group. If the weighted score exceeds a threshold value, then the result is accepted and each mapper that computed that result has its weighted reputation score increased. Those mappers that computed results that fell in other result groups are assigned an error score. When the error score exceeds a certain value, the worker is marked as malicious and no further tasks will be scheduled on it.

Huang et al. [166] detail a result verification method for particular document processing applications such as inverted indexing. They perform a watermarking injection procedure on randomly selected documents from the input collection. For those documents that are selected certain words in the document are substituted. Only the verifier knows which documents were selected and which words were substituted. The verifier then computes the results of the map task for the chosen documents. When mappers complete their tasks, the verifier checks to see whether one of the watermarked documents is contained in those processed by mapper. If so, it compares the mapper’s result with its own. If they do match, it is decided that the mapper is cheating.

Ren and Tang [282] present a scheme that is very similar to VIAF [337]. They additionally implement a timeout mechanism that allows the master and verifier to determine if a worker is being maliciously slowed.

Wang [333] describes a result integrity mechanism for summation form machine learning algorithms implemented on MapReduce. This solution uses a combination of majority voting and watermarking (by artificial data injection) to detect collusion.

Wang and Shen [336] propose an audit-based method for ensuring the integrity of the MR computation. They insert logging statements into the MapReduce programs. These execution traces are recorded by their system. Their system audits the execution of map and reduce phases by computing a control flow graph and pre and post conditions for the user-defined functions. It then uses the execution traces to simulate the execution and verify that execution was consistent with the computed graph.

Chiesa et al. [84] gives a method of result verification using zero-knowledge proofs. In this system, a separate proving cluster generates ZKPs of the MapReduce final results. These proofs are verified by the client who receives both the results and
the associated proofs. The proofs are implemented as Merkle hash trees [224].

Wei et al. [345] detail a verification scheme for computation in the cloud which combines Commitment-Based Sampling (CBS) [115] with designated verifier signatures [167, 185, 361].

3.3.3 Other Approaches

In this section, we discuss other approaches to providing security for MapReduce. First, we discuss the current state of security in MapReduce frameworks. In particular, we critically evaluate the Hadoop security design [251]. Then we then look at recent research in areas of MapReduce security unrelated to result verification or computing over encrypted data.

Apache Hadoop Security Design

The current state of security in MapReduce application frameworks is unclear. The original MapReduce paper [105] and the majority of MapReduce implementations, such as [6, 126, 247, 314], do not provide an architecture for security of MapReduce components. This is to be expected, the focus of these projects is the performance of the MapReduce application framework rather than its security. They assume that security is provided by means which are independent of the framework, such as host operating system-based authentication and access control systems on worker nodes. However, such systems do not authenticate messages sent between worker nodes, making it easy for an attacker present within the network to forge messages. Such a system places a burden on the network gateway to prevent unauthorised network access.

Apache Hadoop is a notable exception in that a security architecture has been provided for the Apache Hadoop MapReduce framework. O’Malley et al. discuss the authentication and authorisation architecture design in [102, 251]. Prior to Hadoop 1.0.0, no security measures over network isolation were provided to protect an application from unauthorised access and this is still the default mode of operation. O’Malley et al. have implemented a design which uses Kerberos [246] to authenticate requests between key participants, such as the servers and users, but which uses DIGEST-MD5 [131] to authenticate other job related requests.
Access to Hadoop services is performed using remote procedure calls (RPC) using Hadoop’s RPC library. The security design adds Simple Authentication and Security Layer (SASL) support to the RPC library. Two security mechanisms are provided using SASL: Kerberos (via Generic Security Service Application Programming Interface (GSSAPI) and DIGEST-MD5. For Kerberos security, the client obtains a service ticket for the given service and mutual security is provided using SASL/GSSAPI. If the client and server share a secret, they can use SASL/DIGEST-MD5 to authenticate to each other. SASL/DIGEST-MD5 is the basis for service to service authentication using Hadoop Distributed File System (HDFS) delegation tokens and MapReduce job tokens as explained below. In addition, authentication by Kerberos over HTTP is provided by a Java SPNEGO library.

To aid discussion of the Hadoop security design, it should be noted that the Hadoop MapReduce design is very similar to the generic model presented in section 4.2. The generic model’s master and worker daemons correspond to Hadoop’s JobTracker and TaskTracker respectively. Similarly, the Hadoop Distributed File System (HDFS) has two components, the NameNode and DataNode, which correspond to the DFS catalogue server and storage server. The major difference between Hadoop and the generic model is that, in Hadoop, mapper and reducers request work indirectly via the TaskTracker on their parent node rather than from the JobTracker directly. The TaskTracker receives work allocations from the JobTracker as part of the heartbeat response. It then apportions the work to mappers and reducers. The generic model also has similarities to the Hadoop security design. The generic security service corresponds to Kerberos. Additionally, the authorisation tickets in the generic model correspond to HDFS delegation tokens and MapReduce job tokens and block tickets correspond to HDFS block access tokens.

In the Hadoop security design, Kerberos is used to authenticate a user to the JobTracker and HDFS NameNode. It is also used to authenticate the JobTracker to the HDFS NameNode, the TaskTracker to the JobTracker, and the HDFS DataNodes to the NameNode. Each of the Hadoop daemons on a cluster node has a unique Kerberos principal. Some services, such as the Oozie workflow scheduler, act as proxies for user requests to Hadoop services. These services have a principal that is configured in HDFS and MapReduce as a super-user, and
3.3. RELATED WORKS

is trusted to act as other users. When a super-user makes an RPC connection, the server checks that the user is able to use the service, the super-user can act on the user’s behalf, and that the request originates from an IP address associated with the super-user service.

HDFS file access by TaskTrackers and map and reduce tasks is authenticated using delegation tokens and block access tokens. Delegation tokens are used in DIGEST-MD5 authentication to a NameNode whereas block access tokens are used to authenticate entities to a DataNode. A user may request a HDFS delegation token after initial authentication to a NameNode using Kerberos and it is used to authenticate job tasks to the NameNode when accessing files. A delegation token acts as a shared secret between the user and a NameNode and all map and reduce tasks of the job will use this token for authentication to the particular NameNode. A delegation token has the following form: 

\{TokenID, TokenAuthenticator\}; where TokenID is \{ownerID, renewerID, issueDate, maxDate, sequenceNumber\} and TokenAuthenticator is HMAC-SHA1(masterKey, TokenID). The master key is a randomly generated 160-bit secret used to create all delegation tokens for that NameNode. The master key is retained, in case the NameNode needs to restart. The NameNode keeps tokens in memory and associates each token with an expiry date after which it will cease to be valid and is deleted from memory. The maximum date specified in the token is a time up to which the token can be renewed even if it has expired. A token will also be deleted if it is cancelled by a user or renewer. The master key is periodically updated every 24 hours and any expired keys will be retained for 7 days so that unexpired tokens may be verified. To authenticate using a delegation token, a HDFS client sends the TokenID to the NameNode. The NameNode checks that the token has not expired and uses the TokenID and the masterKey to compute the TokenAuthenticator. The client uses SASL/DIGEST-MD5 to authenticate using its own copy of the TokenAuthenticator as the password for the DIGEST-MD5 protocol. A request to obtain a delegation token from a NameNode requires authentication via Kerberos. When requesting a delegation token, the user must also specify a Kerberos principal as a token renewer. The token renewer must authenticate using Kerberos in order to renew the token. Renewing a token extends its validity period rather than issuing a new token. For a MapReduce job to use the token, the JobTracker principal must be designated as the token renewer. When the client submits the job, it passes the delegation tokens
for HDFS NameNodes to the JobTracker. The tokens are stored in the \( (key, value) \) pair store in the JobTracker’s system directory. The delegation tokens are distributed to TaskTrackers by RPC.

HDFS block access tokens are used to authenticate a client to a DataNode and to make authorisation decisions about file block access. Previously, Hadoop offered no access control for block access making it possible for anyone to read a block by supplying its block ID. Authorisation decisions are made at the NameNode only, which keeps the permission sets for blocks as part of its metadata. However, the DataNodes that host the blocks of data are unaware of the authorisation decision. The block access token is intended to enforce this decision on the DataNode. When the authorisation decision is made, the NameNode sends a block access token along with the relevant block ID and DataNodes for the given block back to the HDFS client. The HDFS client presents the block access token along with the block ID to the DataNode which verifies that the client is permitted to access the block. Block access tokens are constructed similarly to delegation tokens. However, block access tokens are intended to be short lived and have no renewal mechanism associated with them. The format is: \{TokenID, TokenAuthenticator\}; where TokenID is \{expirationDate, keyID, ownerID, blockID, accessModes\} and TokenAuthenticator is HMAC-SHA1(key,TokenID). The key is a secret shared between the NameNode and all of its subordinate DataNodes. This 160-bit secret key is randomly chosen by the NameNode at start up and sent to the DataNodes when they register with the NameNode. The key is periodically updated every 10 hours by a key rolling mechanism and the new key is then distributed to the DataNodes as part of the heartbeat reply. Retired keys must be kept as they will be used to verify unexpired tokens. Each key is therefore associated with an expiry date and is not persisted to disk. DataNodes cache keys, so if the NameNode restarts, thereby losing its keys, the DataNodes are still able to validate existing tokens. When the block access token is submitted to the DataNode, the DataNode computes the message authentication code for the token using its copy of the associated key. If this is the same as the supplied token authenticator then the request is authenticated and the specified access modes are applied.

MapReduce tasks are authenticated to TaskTrackers by DIGEST-MD5 authentication using \textit{job tokens}. When a job is initialised, the JobTracker creates the
job token, a random sequence of 20 bytes. This job token is stored in the system
directory and is distributed to TaskTrackers by RPC. The job token is used to
authenticate tasks to the TaskTracker when reporting status or requesting work.
The TaskTracker writes the job token to local disk to a directory visible only to
the job’s owner which is also visible to tasks as they run under the owner’s security context. DIGEST-MD5 authentication is used between task and TaskTracker
using the job token as the password. When a map task finishes, its intermediate data is given to the parent TaskTracker. Reduce tasks fetch this data from
the TaskTracker using HTTP. To prevent unauthorised tasks accessing map task
output, a message authentication code is computed as part of the request using
HMAC-SHA1 with the job token as the key.

One may wonder why the Hadoop security design uses proprietary Hadoop credentials rather than using Kerberos authentication throughout a Hadoop cluster. Page 9 of [251] outlines the reasons. To summarise, they are:

**Performance** The current security model for Hadoop MapReduce requires tasks
to run in the security context of the user who submitted the job. This
means that if Kerberos were deployed to authenticate tasks to other Hadoop services, then a delegated TGT (ticket granting ticket) or a delegated service
ticket would be required. If delegated TGTs were used, then the KDC (key
distribution centre) could easily become a bottleneck.

**Java support for Kerberos delegated service tickets** Java’s GSS-API does
not support delegated service tickets. For portability of code, the Hadoop Security Team chose not to use a third party native Kerberos library to
provide delegated ticket support.

**Credential renewal** For long-running tasks, a delegated TGT or service ticket
would need to be renewed. Using Kerberos, a new ticket would be issued
which would need to be distributed to running tasks. Also, timing becomes
critical as the ticket needs to be renewed before its expiry time.

**Less damage if a credential is compromised** A compromised TGT can be
used to access other services. The damage is greater than a HDFS-only credential.

**Compatible with non-Kerberos security schemes** security mechanisms,
such as SSL, are compatible with the proprietary credentials.

O’Malley et al.’s design has a number of flaws, some of which are discussed in [26]. The use of symmetric encryption in the token design is problematic. Only one job token is created for a job and this is used as the secret key for DIGEST-MD5 authentication between all tasks and TaskTrackers. Similarly, each job has only one delegation token per HDFS NameNode. The TokenAuthenticator of the delegation token is used as the secret key for DIGEST-MD5 authentication to this particular NameNode by all TaskTrackers and tasks. As a result, compromising one TaskTracker will reveal the job token and delegation tokens for the entire job. Similarly, the secret key that is used to create block access tokens is shared between a NameNode and all of its subordinate DataNodes. Compromising one DataNode would reveal this key.

Furthermore, the default method of distributing secrets is unencrypted. The default SASL quality of protection (QoP) for RPC is ‘auth’, authentication only. Therefore, as tokens and keys are distributed by RPC, this call is unencrypted by default. An attacker who monitors traffic in the cluster will be able to intercept these secrets. It should be noted that RPC is encrypted if the SASL QoP is configured to ‘auth-conf’. However, the recommended QoP is ‘auth’ to avoid performance penalties associated with encrypting every RPC.

It should also be noted that delegation and block access tokens are easily transferable. As of Hadoop 1.0.0, the owner ID of a token is not checked [347]. Even if it were, the ownerID is plaintext and a request may easily be forged to contain the owner ID of a stolen token. Furthermore, block access tokens are easily replayable as they are transmitted in entirety, including the TokenAuthenticator, to the DataNode. An attacker sniffing on the network can intercept the token and replay it to access the block.

Some Hadoop implementations use HDFS proxies for server to server bulk transmission of data which are authenticated by IP address only. The security design does not address this scenario and it is easy for an attacker to spoof a proxy IP address, authenticate, and then direct a proxy to transfer HDFS files to an unauthorised recipient.

Finally, it should be noted that DIGEST-MD5 has now been made obsolete as an SASL layer [219] and will not be supported in future revisions of the SASL
standard.

So, if Kerberos entirely replaced the proprietary authentication credentials proposed by O’Malley et al. would this address the problems described above? Unfortunately, Kerberos has several drawbacks. Kerberos is based on the original Needham-Schroeder protocol \[244\]. It uses a trusted third party, the key distribution center (KDC), composed of two servers, an authentication server (AS) and a ticket granting server (TGS). The ticket granting server creates the session keys used to authenticate claimants. The authentication server is responsible for creating an encrypted token, called a ticket granting ticket (TGT), which a claimant uses authenticate itself to the TGS in order to receive a session key.

The first drawback with Kerberos is that the KDC represents a single point of failure. If it is unavailable, then clients are unable to authenticate. This can be mitigated by providing multiple KDCs, as employed in Microsoft Windows Active Directory \[226\]. Such a method requires that the client master secrets are synchronised securely between KDCs. Second, Kerberos relies on time synchronisation between the KDC and client and between clients and services. Otherwise, a ticket will be rendered invalid as a verifier will believe the ticket is replayed. Third, if a client is compromised then the client master secret is compromised and an attacker will be able to impersonate the client. Furthermore, if the KDC is compromised then all client secrets are compromised. Finally, common implementations of Kerberos, such as MIT’s krb5 \[214\] and Active Directory, provide the AS and TGS as components of a single KDC server. If many clients wish to authenticate to many different services, for example, in MapReduce, the reducers collecting intermediate data, then this can create a significant workload on the KDC. For applications where performance is a consideration, such as MapReduce, the KDC can become a bottleneck. This problem could be mitigated by providing the AS and TGS as separate servers and then distributing the TGS request load by providing multiple TGS servers. Another alternative is to offload the majority of the work of authentication onto clients and server as outlined in the proposed Public Key-based Kerberos for Distributed Authentication (PKDA) extension \[303\].

Some of the drawbacks described above are addressed in other protocols. The amended Needham-Schroeder \[243\] and Otway-Rees \[261\] protocols demonstrate
that time synchronisation is not required to avoid replay attacks. Neuman-Stubblebine [245] and Kehne-Schönwälder-Langendörfer [187] allow for re-use of the session key to re-authenticate to the same service. This substantially reduces the load on the authentication server. However, both protocols have been proven vulnerable to the attacks outlined in [168]. Kao-Chow [186] allows re-use of the session key and is impervious to the attacks in [168]. However, this still requires the authentication server to be online. Boyd [62, 63] provides a protocol where fresh session keys are generated using an agreed upon keyed hash function and the original session key. This allows authentication to occur even if the authentication server is offline. However, such a system requires a mechanism for revoking session keys which is not detailed in the paper. Without a revocation system, an attacker would be able to use stale session keys to successfully authenticate.

Finally, Crispo et al. [98] provide a protocol and framework based on an offline trusted third party (TTP) server. In this scheme, every client initially authenticates and registers with the authentication server, is assigned an integer index, $i$, and receives an database of introduction tickets, which contains one entry for every other client $j$ where $j \neq i$. Each entry is a pair of a random symmetric key, $K_{i,j}$, and an introduction ticket, $IC_{i,j}$. The introduction ticket is encrypted with the master key of the $j$th client and it contains the symmetric key, $K_{ij}$. The authentication process is as follows.

\[
\begin{align*}
A \rightarrow B : & i||N_A \\
B \rightarrow A : & j||N_B||IC_{j,i} \\
A \rightarrow B : & E(K_{A,B}, ID_A||N_B)||IC_{i,j} \\
B \rightarrow A : & E(K_{A,B}, ID_B||N_A)
\end{align*}
\]

where $N_A$ and $N_B$ are nonces and $K_{A,B}$ is derived using a hash function, $H$, as $K_{A,B} = H(K_{i,j}||K_{j,i}||N_A||N_B)$. Note that $A$ can decrypt $IC_{j,i}$ to retrieve $K_{j,i}$. Similarly, for $B$. Re-authentication proceeds with each client choosing a new nonce and generating a new shared key. This system does address key revocation using key revocation lists. These lists are published by the authentication server and pushed to several locations. The clients check the lists to verify the freshness of an introduction ticket. However, this protocol addresses the problem of an offline TTP by reintroducing the $n^2$ key problem. If there are $n$ clients, the authentication server must generate $n^2$ master session keys to distribute as introduction.
3.3. RELATED WORKS

Recent Apache projects have attempted to address the shortcomings of the Apache Hadoop security design. Apache Knox [12], provides a gateway application for client interaction with the Hadoop REST APIs. Knox implements several methods for authentication of REST API calls including Kerberos [214], OAuth [159], and SAML [278]. Apache Sentry [15] provides fine-grained role based authorization for data in a Hadoop cluster. Apache Ranger [14] also provides a fine-grained authorisation framework for Hadoop which supports both role based and attribute based access control. However, the Knox and Ranger projects are still in the early stages of development and none of these projects are part of common Hadoop deployments.

Other Approaches

We now discuss other approaches to MapReduce security in the literature. Lin et al. [208, 209] discuss the use of the Shamir threshold secret sharing scheme [297] to allow the cluster to recover from the failure of a Public Key Infrastructure (PKI) server safely. Each map worker is assigned a piece of the PKI private key, a shadow, by the PKI server as an individual secret key. They also devise a group signature scheme based on threshold secret sharing. This scheme is used by the map workers to sign intermediate data. In the event that the PKI server fails then the secret key can be reconstructed by a new master by assembling \( t \) shadows from map workers. This scheme has the advantage that clients do not need to be re-enrolled. Provided revocation lists are published in an accessible location, the reconstruction of the server private key allows the system to return to its state before failure. They do not discuss how the cluster would distinguish between a PKI server that has been legitimately elected and a rogue PKI server. We assume that an authorised PKI server would have to authenticate to \( t \) nodes to retrieve the \( t \) pieces required to reconstruct the PKI secret key. This process could not be authenticated by digital signatures because the PKI is not yet operational.

On a final note, Lin et al.’s scheme has a severe performance overhead. With 40 tasks, a MapReduce job takes twice as long to complete using their mechanism compared to without security.

In some MapReduce applications, the data processed is sensitive and must be
protected from unauthorised access. Various papers discuss adding authorisation systems to the MapReduce distributed file system. In Hamlen et al. [158], they integrate a policy-based authorisation mechanism into Hadoop’s HDFS using Extensible Access Control Markup Language (XACML). To achieve greater flexibility they propose implementing the policy enforcement and decision points as in-lined reference monitors (IRM). These IRMs are binary rewriters that rewrite the bytecode of an untrusted process and inject security guards around untrusted operations. This prevents an attacker circumventing any cluster level access control by launching programs on cluster nodes. Thuraisingham et al. [316] also describe a solution which applies XACML to HDFS for use with Apache Hive and Hadoop. Roy et al. [289] implement a data privacy framework, Airavat, for MapReduce applications. This approach applies data privacy using two methods. The first is to apply mandatory access control to the MapReduce distributed file system. The second method uses differential privacy [119] to prevent leaks via the output of the computation. The performance overhead for Airavat is not insignificant, a MapReduce computation takes 32% longer to complete with Airavat compared to a cluster without Airavat. Each of these methods relies on authentication to prove that parties are those that are authorised to access data. The authentication method used by either method is not discussed. Without an adequate authentication method that makes masquerade attacks unlikely, sophisticated authorisation controls such as these are meaningless. Ulusoy et al. [322] discuss GuardMR, a system for fine-grained access control for MapReduce. Their solution takes authorisation policies described using Object Constraint Language (OCL) and converts them into functions, and Java bytecode, which enforce the policy. Yu et al. [351] describe SEHadoop, an improvement on O’Malley et al.’s design. They provide alternative definitions for the HDFS block and delegation tokens to prevent some of the flaws present in O’Malley et al.’s design. Rather than a single block token shared between the client, a NameNode, and all its subordinate DataNodes, they have implemented separate block tokens for each HDFS DataNode based on a secret key shared between the client and the DataNode. To provide fine-grained access control they divide the delegation token into a parent token shared by the client and the NameNode, and child tokens which are generated by the NameNode. The parent token contains no fine-grained access control but is used to create child tokens which contain the access control list for the requested file. This design addresses the problems with reuse of credentials
3.3. RELATED WORKS

in O’Malley et al.’s design. However, the parent delegation token contains the ID of the parent token and the key used to generate child tokens stored as plaintext. These are the only two items in common between parent and child tokens. Therefore, gaining access to a parent token allows one to create child tokens as required. Shen et al. [300] present their Security Architecture of Private Storage Cloud (SAPSC) which adds role-based access control (RBAC) to HDFS. As an option, AES encryption of files is supported. The AES keys are stored in a hardware trusted platform module (TPM). However, they do not detail this encryption process.

Several papers approach encrypting the MapReduce distributed file system, particularly HDFS. Patil and Venkatesan [266] provide an extension to Hadoop to allow MapReduce to operate on encrypted files in the Hadoop distributed file system (HDFS). The input data is encrypted by their own stream cipher. The job owner is authorised using OAuth 2.0 [159]. The OAuth token is used to generate the bit stream used to XOR with the data stream. The OAuth token is passed to MapReduce worker nodes to allow them to reconstruct the bit stream and thereby decrypt the input files. Although this system prevents snooping on the file system, it does not secure the unencrypted data on the worker nodes. Similarly, Park et al. [265] describe an extension to HDFS that supports AES encryption. Although the performance penalty is minimal (a 7% overhead), this is a solution which must be employed in conjunction with a method which prevents information leakage on the MapReduce worker nodes. Lin et al. [94] describes how to implement the Tahoe least authority file system (LAFS) for use with MapReduce. Tahoe uses AES to encrypt file blocks and public key encryption to protect the AES symmetric key. Lin et al. [207] also propose a scheme where HDFS file blocks are encrypted using AES. The AES key is protected by RSA public key encryption or by a pairing-based encryption [259]. Cohen and Acharya [91] devise a similar system. However, here the RSA private keys are stored in a hardware-based trusted platform module. Jing et al. [175] and also provide an encrypted HDFS based on AES and RSA. Yang et al. [356] propose a similar system where the files are encrypted with DES [239] and the RSA key is encrypted with the International Data Encryption Algorithm (IDEA) [197]. However, there are concerns about the security of both DES and IDEA. DES has now been superseded by AES and IDEA was broken by a meet-in-the-middle attack by Demirci et al.
Madaan and Agrawal implement an encrypted HDFS using a combination of \((t, n)\) threshold secret sharing and fully homomorphic encryption. The system generates an asymmetric key pair for a user along with a unique one-way function. The one-way function is used to retrieve the private key from a secret known only to the user. One of the shares is used to encrypt the user’s unique one-way function. Likewise for the user’s private keys, user’s file catalogue, and the user’s public key store. Madaan and Agrawal do not outline the benefits of this elaborate system nor how FHE is required in its construction.

There are several hardware based solutions. To et al. present their TrustedMR which uses trusted hardware to provide trusted data servers. In this scheme the trusted data servers are responsible for encrypting any data that is confidential and for placing the data on the distributed file system. The MapReduce runtime behaves normally operates on unencrypted data. On encrypted data, the MapReduce workers forward the computation to the trusted data servers and receive the results. Schuster et al. also describe a hardware based method for secure, assured MapReduce computing taking advantage of the Intel Software Guard Extensions (SGX) trusted computing platform. In their scheme, the user-defined map and reduce functions execute in the secure SGX enclave. The insecure MapReduce framework can only access these functions by a limited interface which allows it to supply input key-value pairs and receive output pairs. To allow the user-defined code to be securely distributed to the MapReduce cluster nodes, it is encrypted. The SGX processor negotiates the secret key with the client in order to decrypt the UDFs. Pires et al. also describe an SGX-based system. Li and Jin propose a solution using software defined network (SDN) switches. A controller is responsible for determining and distributing forwarding rules to the Layer 2 switches. With this scheme they segregate worker nodes into groups of varying trust. Dou et al. describe a TPM-based protocol for authentication of the MapReduce framework software components.

Liao and Squicciarini describe a system that captures and manages MapReduce provenance data for the purpose of anomaly detection. The provenance capturer processes worker logs. This provenance data is then analysed for anomalies. Although, this system mainly detects faults in the MapReduce computation, it can also be used to detect cheating workers.
K. Zhang et al. [362] discuss a data tagging based approach for use in hybrid clouds. The input data is partitioned into sensitive and non-sensitive collections and tagged accordingly. The sensitive data is processed in the private cloud, the non-sensitive data in the public cloud. Similar approaches are C. Zhang et al.’s Tagged-MapReduce [360] and Oktay et al.’s SEMROD [254]. In [44], Bissiriou and Zbakh detail their proposal for a TPM-based architecture for Tagged-MapReduce.

Zhao et al. [364] discuss a security framework for G-Hadoop [334], a MapReduce implementation for large-scale distributed computing across several clusters. As G-Hadoop is mainly intended for scientific computation, the focus is on isolating user applications and data from other applications and to enforce authorisation for resources in each cluster. They add a single-sign-on authentication system and access control to G-Hadoop.

Project Rhino [171] provides a comprehensive security framework for Hadoop. Rhino provides centralised authentication, authorisation and auditing services. In addition, Rhino also provides encrypted communication support for Hadoop and an encrypted file system for Hadoop clusters.

### 3.4 Chapter Summary

This chapter has performed an extensive review of the security literature with a view to identifying the best methods applicable to cloud MapReduce applications.
Chapter 4

Threat Analysis of MapReduce Computation

The material in this chapter was presented in abridged form at the 8th International Workshop on Security and High Performance Computing Systems (SHPCS 2013) [122].

4.1 Introduction

In this chapter we provide a more detailed analysis of a MapReduce computation than the one given in section 2.3. We trace the execution flow of a generic model of MapReduce job from initiation to completion. This generic model is derived from a number of MapReduce architectures, in particular Google MapReduce [105] and Hadoop MapReduce [347]. In particular, the security measures outlined in this model, the authentication service and authorisation tickets, are directly inspired from Hadoop’s security architecture. We have included these features to demonstrate that the Hadoop security design is vulnerable to a number of attacks. We provide a security threat analysis of this execution flow. Finally, we discuss how these threats may be countered.
4.2 A Generic Model of a MapReduce Job Execution

To illustrate the MapReduce job flow process, we follow the execution of an example cloud MapReduce application. It is a word count application distributed across four nodes, a master node and three worker nodes. The application counts words appearing in a collection of text documents.

4.2.1 Physical Architecture

For completeness, we assume that the MapReduce application is physically deployed as virtual machines (VMs) in a public cloud (see Figure 4.1). The case where the application is deployed via containers is identical to that as described in section 4.2.2 In our model, one VM is a master node and the other VMs are worker nodes. These VMs are hosted on physical virtualisation hosts within the cloud infrastructure. As shown in the figure, a VM may share tenancy on a virtualisation host with other VMs. It may be part of the MapReduce application or related to another customer’s application. The VMs for each customer application placed on the same virtualisation host are isolated from each other.
using the dedicated virtual switch for each application. A VM can communicate with another VM on the same virtualisation host via a virtual switch. The switch accesses the physical network via the virtualisation host’s kernel, known as a hypervisor. The hypervisor puts data frames passed down from the switch on to its network interface.

A client application is responsible for configuring, starting and monitoring the MapReduce job. In an IaaS MapReduce implementation, the client will manage the MapReduce architecture directly by accessing the master node VM. In a PaaS implementation, the client will connect to a management interface which uses cloud middleware to manage the MapReduce architecture. The interactions between the client and the cloud are performed via the Internet.

4.2.2 MapReduce Architecture

In the MapReduce architecture, we have four MapReduce cluster nodes: a master node and three worker nodes (see Figure 4.2). The master node executes a single master daemon. The master daemon is responsible for coordinating the execution of the application on the worker nodes. Similarly, each worker node executes a single worker daemon. The worker daemon is responsible for launching and managing mapper and reducer child daemons on the node. A mapper executes a single map task. Likewise, a reducer executes a single reduce task. In our example, each worker daemon will manage one mapper daemon and one reducer daemon. We denote each daemon process to be a component of the distributed MapReduce application. In addition, we designate the collection of components that form our MapReduce application to be a framework. We denote the communication paths between components to be channels.

In addition to the MapReduce framework components, we have nodes that host an authentication server for entity authentication, nodes that host vital networking services (not shown on Figure 4.2), such as DNS or DHCP, and a distributed file system (DFS) which is hosted on its own cluster. The distributed file system is a block oriented file system where each file is stored as a number of fixed size data blocks distributed across the DFS cluster. The DFS has two independent servers, a catalogue server, that is responsible for maintaining the file system directory tree along with the locations of each block associated with a file, and a storage
server, that is responsible for hosting the individual file blocks. Each node of the DFS cluster hosts a single server. There is one catalogue server per DFS cluster and multiple storage servers.
4.2.3 Logical Architecture

The logical flow of the application is shown in Figure 4.3. The application processes input files stored on the distributed file system. Our example application processes three files. The input files are logically divided into input splits where there is at least one input split per mapper process. In the example word count application described as the beginning of section 4.2, each of the input splits is an individual document. Each input split is assigned to a mapper by the master. A mapper processes its input split and writes the output locally. The local output is divided into a number of data sets called partitions. The number of data partitions per mapper is the same as the number of reducers. Each partition divides a mapper’s output according to some specification. In the example, there are three partitions which are divided according to the alphabetical order of the counted word. Each reducer is assigned a partition index to process. Once the mappers have completed, the reducer will read the partition with its assigned index from each mapper. This data is combined and sorted, and then processed. The reducer’s output is written to a file in a folder on the distributed file system. The output folder will have one file for each reducer.

Figure 4.3: Logical architecture of a MapReduce application (adapted from [105])
4.2.4 Job Execution Flow

The application executes as follows. First, the framework components are launched. A master daemon process is launched on one cluster node, the master node, and a worker daemon is launched on each of the worker nodes. Before a worker daemon can start launching mappers and reducers it must obtain admission to the MapReduce cluster (Figure 4.4). The worker daemon first authenticates itself to the master.

The component authentication process (Figure 4.5) is identical for all components so we will describe it once here. A component, for example, the worker daemon, makes an authentication request to the authentication server using its credential (Figure 4.5(A1)). If the authentication server verifies the credential it sends a master ticket in its response (Figure 4.5(A2)). The master ticket is used to obtain service tickets to authenticate the component to remote resources, such as the master daemon. It now makes a request for a service ticket for the resource it wishes to access (Figure 4.5(B1)). It sends its ticket and the resource address. The authentication server verifies the ticket and, if successful, sends a service ticket to the component (Figure 4.5(B2)). Once a service ticket has been obtained, it can be used to authenticate a request for the specified resource by the entity (Figure 4.5(C1, C2)).

Therefore, once the worker daemon has obtained a service ticket for the master, it requests admission to the MapReduce cluster of the master daemon (Figure 4.4).
A1), sending its node identifier and the service ticket. The master verifies whether it is authorised to participate in the cluster and sends a response indicating whether the worker has been admitted or not (Figure 4.4(A2)).

![Figure 4.6: Job submission process](image)

Once all the worker nodes have been admitted, the client writes the input files and job program files to the distributed file system (DFS) (Figure 4.6(A1)). In our example, the input files are text documents and the job program files are scripts or Java class files that contain the map and reduce functions in a format that can be invoked by the mappers and reducers respectively. The client must first obtain a service ticket for the DFS catalogue server as outlined above. The client program then submits a write request to the DFS catalogue server, that includes the requested operation (write), the file system namespace names of the files and the service ticket (Figure 4.7(A1)). The DFS catalogue server will verify that the client is authorised to write to the file system at the specified file path. The catalogue server will send a response to indicate that it accepts or denies the write request (Figure 4.7(A2)). If it accepts the write request, it includes the address of the DFS storage server that will host the new block of data, the file system

![Figure 4.7: DFS write process in detail](image)
identifier of the new block, and a block ticket. This block ticket both authenticates the client to the storage server and also acts as a capability to specify what access rights the client has for the specified block. The client now requests to write the data to the block on the specified DFS storage server (Figure 4.7(B1)). It includes the block identifier and the block ticket in its request. The DFS storage server unpacks the block ticket and verifies the request. It sends an acknowledgement that the write request has been accepted or denied (Figure 4.7(B2)). If the write request has been accepted the client streams the data to the storage server which is written to the block (Figure 4.6(C1)). When the write process has completed, the DFS storage server sends an acknowledgement to the client indicating whether the operation completed successfully or not (Figures 4.7(C2), 4.6(B2)). If there is more data to write than a single block, the write process repeats from the beginning until all data has been successfully written.

Once the input and program files have been written on the DFS, the client calculates the input splits and submits the job details to the master by making a job submission request (Figure 4.6(B1)). The request contains the DFS namespace names of the input and program files, the calculated input splits and a job configuration. The master verifies that the client is authorised to submit a new job. If not, the master sends an acknowledgement to the client that the job has been rejected. Otherwise, the master verifies that the input and program files exist and then initialises the job. It generates an identifier for the job and identifiers for the map and reduce tasks specified in the job configuration. These identifiers are stored in its metadata. The master sends an acknowledgement to the client to indicate that the job has been successfully submitted (Figure 4.6(B2)). This acknowledgement contains the job identifier. The client needs the job identifier to retrieve status information about the job.

During the job execution, each worker daemon periodically sends a heartbeat message to the master daemon (Figure 4.8(A1)) to inform the master that the node is still operational. The heartbeat message may also be used as a data channel for the master to communicate data with the worker node. The message contains the worker node’s identifier and the service ticket for the master. For each message received, the master sends a response to indicate how many mappers and reducers to launch on the node. If map and reduce tasks are currently executing on the node, then the master’s response may also inform the worker if any of the
tasks should be killed. The response from the master includes the job identifier, identifiers for each map and reduce task to be launched or killed, and the filenames of the job program files (Figure 4.8(A2)).

Before launching mappers and/or reducers on the node, the worker daemon needs to first fetch the job program files from the DFS and cache them locally. For this, the worker daemon makes a read request to the DFS (Figure 4.8(B1)). The worker daemon obtains a service ticket for the DFS catalogue server as outlined above. It then sends a request to read the specified files to the DFS catalogue server (Figure 4.9(A1)). The request contains the DFS namespace names of the job program files and the service ticket. The catalogue server verifies the service ticket and also verifies whether the worker daemon is authorised to read the specified files. It sends a response to indicate whether file access has been
4.2. GENERIC MODEL

If the read request has been granted, this response also includes the addresses of the DFS storage servers that host the blocks that comprise the files, the identifier for each block, and block tickets for each block. The worker daemon then requests to read the specified blocks from each of the specified DFS storage servers (Figure 4.9(B1)). This request contains the identifier of the requested block and the block ticket. The DFS storage server unpacks the ticket and verifies that the worker daemon should have access to the file. If access is granted it sends the block, otherwise it sends a notification that the read request has been refused (Figure 4.9(B2)). When all blocks have been successfully fetched, the worker daemon reconstructs the files and caches them (Figure 4.8(B2)). Once this is done, the worker daemon launches the mappers and/or reducers (Figure 4.8(L)). For each mapper or reducer it launches, it supplies the following parameters: the node identifier; the job identifier; the identifier for the map or reduce task; and the location of any required program files in the cache.

Each of the mappers now request work from the master (Figure 4.10(A1)). The mapper sends a request message that includes the job identifier, task identifier and node identifier. The master schedules map tasks to be executed by each
respective requester. Its response includes the input split it has assigned to the mapper and the location of the input file to read and an authorisation ticket for the DFS catalogue server (Figure 4.10(A2)). The authorisation ticket is used in lieu of a service ticket when the mapper requests the input file. This ticket not only authenticates the mapper to the catalogue server but also indicates which files the master has authorised it to read.

The mapper reads its assigned input split from the DFS (Figure 4.10(B1)). It sends a read request to the DFS. This request includes the namespace name of the input file, the task identifier of the map task, the job identifier of the parent job, and the authorisation ticket supplied by the master. The DFS read process is identical to the description above. The mapper fetches the input split from the DFS and parses the input split as key-value pairs (Figure 4.10(B2)). In this example, each mapper reads one line of the input document. The mapper now invokes the user-defined map function on each key-value pair. The partitioning function separates the output into three sets. Each result set is written to a separate location on local storage (Figure 4.10(I)). The mapper now informs the master that it has completed successfully or the computation has failed (Figure 4.10(C1)). Its notification includes the node identifier, the task identifier for the mapper, and the job identifier. The master responds with an acknowledgement (Figure 4.10(C2)).

The reducers periodically poll the master for work. Each reducer sends a request message that includes the job identifier, task identifier and node identifier (Figure 4.11(A1)). When all mappers have notified the master that they have successfully completed, the master chooses a partition for the reducer to process. It responds to the polled request by sending the partition index, an authorisation ticket for the DFS catalogue server and authorisation tickets for each mapper (Figure 4.11(A2)). The authorisation ticket for the catalogue server is identical to that supplied to a mapper as described above. It is used in lieu of a service ticket when the reducer requests to write its output to DFS. The authorisation tickets for the mappers serve as service tickets as well as indicating that the ticket holder has been authorised to read the specified partition.

The reducer reads the key-value pairs from the specified partition on each mapper (Figure 4.11(B1)). The reducer makes a request to the mapper to read the
intermediate data partition from that mapper. This request includes the partition index, the task identifier for the reduce task, the job identifier for the parent job, the node identifier, and the authorisation ticket supplied by the master. The mapper validates the authorisation ticket and verifies if the reducer is authorised to read the partition. If this is successful, the mapper sends the requested key-value pairs to the reducer (Figure 4.11 B2). The reducer sorts the key-value pairs received from all mappers by key. Where pairs have the same key, they are combined into a single pair of the key and a list of values. Each reducer writes its output to a file on the DFS (Figure 4.11 C1). It makes a write request to the DFS. This request contains the reducer’s task identifier, the job identifier, the node identifier, the namespace name of the output file, and the authorisation ticket supplied by the master. The DFS write process is outlined above. Once the output has been successfully written to the DFS file, the reducer now notifies the master that it has successfully completed the computation or, if the write process failed, that it has failed (Figure 4.11 D1)). This notification includes the node identifier, the task identifier for the reducer and the job identifier. The master responds with an acknowledgement (Figure 4.11 D2)).
Once all reducers have completed, the master sends a notification of job completion to the client (Figure 4.12(1)).

4.3 A Threat Analysis of the Job Execution Flow Model

In this section we identify threats to each request-response interaction between the MapReduce components. We concentrate on threats that relate to peer-entity and message origin authentication. Our two main threats are disruption of the computation and eavesdropping on the computation.

We classify attacks on the MapReduce application as masquerade, replay, message modification, and denial of service. A masquerade attacks occurs when an attacker impersonates an authorised identity. With regard to the model discussed above, this may occur as a result of a stolen credential, by bypassing the authentication mechanism or by fabricating message headers to make an injected message appear to have originated from a trusted entity. Replay attacks occur where an attacker intercepts a message and re-sends the message or parts of the message to the intended recipient. Message modification occurs when a message between two components is altered to an attacker’s advantage. This is achieved by intercepting, altering and then forwarding on an existing message. Finally, denial of service occurs where an attacker actively disrupts the operation of a component so that it is unable to process genuine requests or disrupts a message channel so that messages cannot be delivered. A denial of service can occur when an attacker floods a component or channel with messages so that resources are saturated on components or networking devices and new messages cannot be serviced as with “smurf” attacks [71] and SYN floods [124]. An alternative attack on components is to send overlarge messages which overrun message buffers and cause the software component to crash, such as teardrop attacks [70].
4.3. A THREAT ANALYSIS OF THE JOB EXECUTION FLOW MODEL

In the following paragraphs, we identify the following messages that are vulnerable to each of the attacks and describe the consequences of a successful attack for that message.

4.3.1 Masquerade attacks on the authentication service

The initial authentication request (Figure 4.5(A1)) authenticates an entity by sending its credential. An attacker that has captured a credential, for example, a password or digital certificate, would be able to forge a request and receive a valid master ticket. This master ticket could then be used to obtain service tickets for other resources.

4.3.2 Masquerade attacks on other requests and responses

The following request and response messages can be impersonated by fabricating a message in one of two ways. The message can be forged to include identifiers for an authorised node, job or task as appropriate. Alternatively, message headers may be forged using the source address of a trusted node. In either case, the message appears to have originated with an authorised entity. We describe the effects of each attack below.

Work allocation request

Mappers and reducers request a task to execute from the master (Figures 4.10(A1), 4.11(A1)). An impersonated request would allow an attacker to gain an authorisation token for an input file on the DFS or for partitioned data on mappers depending on whether a map or reduce work request was forged.

Heartbeat message response

An impersonated heartbeat response (Figure 4.8(A2)) could be used to direct the worker daemon to cache malicious program code of the attacker’s choosing, to make the node unusable by instructing it to launch more processes than the node can reliably manage, or to delay the job execution time by instructing the
node to launch fewer processes. Such code could disrupt the computation or gain control of the host.

Work allocation response

An impersonated response from the master to a work request (Figures 4.10(A2), 4.11(A2)) would allow an attacker to disrupt the job computation by specifying non-existent input files for a mapper to process or an invalid partition index for a reducer to process.

Intermediate data response

A masqueraded response from the mapper (Figure 4.11(B2)) could be used to misinform the mapper that access to a partition has been denied. Without access to all the partition records from the mappers, the reducer is not able to complete its computation.

DFS catalogue server read response

Impersonated read responses from the DFS catalogue server (Figure 4.9(A2)) could be used to misinform the requester that file access has been denied or to direct the requester to a DFS storage location of the attacker’s choosing. This might allow an attacker to halt the job execution by denying access to the user-defined program files or input files. In the former case, a worker daemon will not be able to launch mappers or reducers. In the latter case, a mapper is not able to execute its task if it has no access to its input split. A forged response could also allow an attacker to specify files to be read by the requester. In the case of the worker daemon fetching the user-defined program files from the DFS (Figure 4.8(B2)) such an attack could direct the worker daemon to fetch malicious program files instead.

DFS storage server read response

Similar to a fabricated response from the catalogue server, a forged response from the storage server (Figure 4.9(B2)) could be used to indicate that the read request
has been denied. This could be used to disrupt the computation by denying access to input data. Alternatively, a crafted response could be used to send a different block of data to the requester. This could be used to send malicious code to be executed on a worker daemon (see Figure 4.8(B2)) or false data to be processed by a mapper (see Figure 4.10(B2)).

**DFS catalogue server write response**

An impersonated acknowledgement (Figure 4.7(A2)) could be crafted to misinform the requester that write access has been denied to the DFS. When this is the client writing files to the DFS, such an attack could be used to prevent the job being submitted by denying the client the ability to write the raw data to the DFS. In the case of the reducer attempting to write its output to DFS (Figure 4.11(C2)), it will be unable to successfully complete its computation as it is unable to write its output to the specified location.

**DFS storage server write response**

A forged response from the storage server (Figure 4.7(B2)) could be also used to deceive a requester that a write operation has been denied. This could be used to disrupt the MapReduce computation by denying a client from writing a data block of an input file to DFS or a reducer from writing its output to the block. Alternatively, it might be used to misinform the requester that the write request has been accepted. This could be used to deceive the client into attempting to write the data and allowing an attacker to intercept it as it is transmitted between the DFS client and the storage server.

**Task completion notification**

On task completion, a mapper or reducer sends a notification message to the master (Figure 4.10(C1), 4.11(D1)). A crafted message could be used to inform the master that a task had failed while it was still executing. This would delay the overall execution time of the computation by forcing the task to be re-executed. Alternatively, a notification could be forged that indicated a failed task had successfully completed. If it was a mapper that had failed, then this
might disrupt the computation as the mapper may have written no intermediate data or a partial results set before failing.

**Write completion notification**

A DFS client attempting to write a block of data to a DFS block storage server receives an acknowledgement to indicate whether the write operation has completed successfully or not (Figure 4.7(C1)). A forged notification could be used to deceive the client that data has been written when it has not in order to disrupt the job execution. Alternatively, it could falsely notify the client that the operation has failed so that the block must be rewritten. This might be used to capture the data as it is transmitted or to increase the overall execution time of the job.

**4.3.3 Attacks through message modification**

**Attacks by modifying requests**

A message modification attack can be leveraged in MapReduce by altering the components of a message not associated with authentication. The components of the message that are used to authenticate the sender are left unaltered so that the message will be accepted by the recipient as genuine. With the exception of an authentication request, the masquerade attacks described in section 4.3.2 may also be achieved by modifying the message headers of genuine messages. In addition to those, we also identify the following additional threats as a result of message manipulation. The DFS read and write requests to the catalogue server (Figures 4.9(A1), 4.7(A1)) can be altered to provide a new message source address. The response from the DFS will be sent to the attacker’s host rather than the original requester. The attacker will receive block tickets which allow access to the file stored on the DFS. Indeed, the same outcome may be achieved for a data block only by similarly modifying a request to a DFS storage server (Figures 4.9(A1), 4.7(A1)). A valid job submission request (Figure 4.6(B1)) might also be altered for malicious intent. Altering the job configuration may change the job execution behaviour to an attacker’s advantage, such as reducing the number of mappers and reducers in order to increase the job execution time. Such an
4.3. A THREAT ANALYSIS OF THE JOB EXECUTION FLOW MODEL

Alteration could allow an attacker more time to perform brute force attacks on cluster nodes. In addition, if an attacker is able to modify the job program files specified in the request then malicious program code can be substituted instead.

Attacks by modifying responses

As noted previously, the masquerade attacks outlined in section 4.3.2 may also be achieved by modifying a genuine response. In this case, the message is altered so that it is identical to the masqueraded response discussed. For brevity, we do not repeat this here.

4.3.4 Attacks by replaying authentication tickets

Replay of a master ticket

If a master ticket (Figure 4.5(A2)) is intercepted, it may be possible to use it to request service tickets (Figure 4.5(B1)). Obtaining a service ticket would allow an attacker access to resources such as the distributed file system.

Replay of a service ticket

The job submission request (Figure 4.4(C1)) specifies the user defined programs that will be executed as part of the job. If this message can be crafted using a replayed service ticket, then the request would be authenticated and accepted. It would allow an attacker to specify arbitrary program code to be executed on the cluster. Similarly, a worker daemon admission request (Figure 4.4(A1)) could be crafted that contained a replayed service ticket in order to admit a malicious daemon to the MapReduce cluster. If a service ticket for the DFS catalogue server can be replayed, then an attacker will be able to access all the files associated with the original ticket holder.
Replay of an authorisation ticket

The master daemon issues authorisation tickets (Figure 4.10(A2), Figure 4.11(A2)) to the mappers and reducers to allow them access files on the DFS (Figure 4.10(B1), Figure 4.11(C1)) and, in the case of the reducer, to access an intermediate data partition on a mapper (Figure 4.11(B1)). If the authorisation ticket is replayable, then the ticket could be intercepted by monitoring traffic. Once an attacker has the ticket he can craft a new request for the data, including the ticket. This request will be authorised by virtue of inclusion of the ticket.

Replay of a block ticket

The requests to read and write blocks on DFS (Figure 4.9(A2), Figure 4.7(A2)) is authenticated using a block ticket. Again, if the block ticket can be replayed then it may be used to authenticate further requests to read the same data block.

4.3.5 Attacks by replaying messages

If a replayed message cannot be detected by the framework this can be used to an attacker’s advantage. Requests to the authentication service (Figure 4.5(A1,B1)) might be replayed in order to intercept the response. This could be used to capture tickets for later use or to invalidate a previously issued ticket. Requests to the DFS catalogue (Figures 4.9(A1), 4.7(A1)) server might be replayed to gain a block ticket. A read request to a storage server might be replayed to force the storage server to send the block again so that it may be intercepted. A write request might be replayed in order to overwrite the block with false data. The work requests from mappers and reducers (Figures 4.10(A1), 4.11(A1)) might be replayed with the purpose of obtaining authorisation tickets for DFS or intermediate data.
4.3.6 Denial of Service attacks

Denial of Service attacks on MapReduce components

We have two critical servers that will cause the execution of the MapReduce job to fail if subjected to a denial of service attack. The master daemon is responsible for scheduling and coordinating execution of the MapReduce application on the worker nodes. The master allocates work to mappers and reducers and also issues authorisation tickets which allow mappers and reducers access to read and write data. A denial of service attack on the master will halt the computation. Similarly, the authentication service is a critical component as it issues service tickets which are used to authenticate components to the distributed file system. Without access to the user defined program files located on the DFS, the mappers and reducers cannot process the data.

Denial of message service attack on node daemon heartbeat messages

If a node heartbeat message is dropped or delayed longer than the heartbeat timeout then the master will assume that the node has failed. It will not schedule future computation on the node. An attacker can reduce available computational resources in the cluster by deleting or delaying heartbeat messages and, in turn, prolong the execution of the job.

4.3.7 Privilege Escalation Attacks

A MapReduce cluster may run multiple jobs concurrently. Also, a cloud MapReduce application may share the same physical hardware with other virtual applications. Where multiple MapReduce jobs are able to run in a cluster, it is possible for an attacker to create a job with the purpose of disrupting or controlling the tasks of another job. A MapReduce job allows for user-defined code to be executed as part of the mappers or reducers on worker nodes. If an attacker is able to launch a job of his own on the cluster, they may be able to execute code written to exploit possible vulnerabilities in the worker node Java Virtual Machine (JVM) or operating system. Such exploitation could lead to the compromise of the worker node. Once the node is compromised, the attacker may
access the input data accessible by the mappers running on the node as well as the intermediate data generated by those mappers. Similarly, gaining control of the reducers on a worker node may allow the attacker to access intermediate data partitions on other worker nodes.

In the case of a cloud MapReduce application, the customer software running on the VMs (see Figure 4.1) is only isolated from other customer applications by the virtualisation host’s hypervisor. If the hypervisor is compromised then all the VMs running on the host can be controlled by an attacker. A hypervisor presents an additional attack target for the virtualisation host. The hypervisor management interface also exposes it to network attacks [325]. Similarly, the virtual switches and simulated hardware drivers, which interact with the hypervisor, expose the host to side channel attacks [349]. Furthermore, two popular virtualisation products, Citrix’s Xen and Microsoft’s Hyper-V, use a highly privileged VM to manage physical hardware access and control the hypervisor. Compromising this VM is equivalent to compromising the hypervisor itself. Xen is used in many cloud implementations, such as Amazon’s Elastic Compute Cloud (EC2) [47], and Hyper-V is used as the basis for Microsoft’s Azure platform.

In addition to the threats posed by the hypervisor, an IaaS provider allows customers to configure VMs as they wish, so an unpatched operating system may be installed on a VM. Also, the cloud provider usually makes a repository of pre-configured VMs available for customers to use. Customers may add VMs into the repository. An attacker may upload a trojanised VM to the repository which is then used as part of a customer application [344]. Finally, cloud VMs have public IP addresses by default which make it possible for an attacker to “sniff” on ports to discover and exploit vulnerabilities in the VM guest operating system and other software hosted.

Where the cloud provider employs containerisation (see section 2.2.3), similar attacks may be employed to allow an attacker to escape the bounds of the application sandbox [320, 350]. These techniques exploit flaws in the host operating system.
4.3.8 Malicious Privileged User

In the case of MapReduce deployed in the cloud, large portions of the MapReduce infrastructure are not under the sole administrative control of the cloud service user. In addition to the system administrators employed by the cloud service user, there are the administrators employed by the cloud service provider. In either case, these individuals are afforded considerable privileges so that they can maintain the application. Should such an individual decide to abuse these privileges, they are easily able to pass authentication and authorisation checks, as they are a highly privileged authorised user.

4.4 Further Discussion

The threat analysis in section 4.3 clearly shows a number of concerns. The first is that messages have originated with an attacker rather than the purported sender. This is a threat we can counter if we are able to authenticate the origin of a message, that is, to prove that the purported sender is the actual sender. The second is that a message has been maliciously altered in transit from sender to receiver. This is a threat that can be avoided if we can verify that the contents of a message received have not changed from the message sent. The third concern is that an attacker re-sends legitimate messages in order to obtain a response. This threat may be countered if a replayed message can be detected. The fourth concern is that key services, such as the MapReduce master, are made unavailable as a result of a denial of service attack. This threat may be mitigated if a service is able to distinguish between legitimate and malicious requests. The fifth concern relates to the authentication methods employed in this generic model. Various authentication tickets may be replayed and, hence, authenticate a malicious request. Again, if a replayed ticket can be detected then this threat is allayed. The authentication service itself is vulnerable to an attacker using stolen credentials to impersonate as an authorised user. Where users configure and manage the MapReduce application from a host outside the cloud this can be a legitimate concern. Trojan horse viruses can easily compromise a user’s host and steal credentials. The final concern is that in the event of a successful privilege escalation attack or if a privileged user, such as a system administrator, behaves
maliciously, then they are able to subvert any security controls we might place on the MapReduce cluster.

4.5 Securing MapReduce Jobs in the Cloud

In this section, we analyse the various security methods presented in chapter 3 with regard to their suitability for cloud MapReduce applications and with regard to the threat analysis presented above. Section 1.3 outlines the key requirements of an ideal security solution for cloud MapReduce applications. These are as follows.

**Scalability** The security solution should scale easily as more nodes are added to a MapReduce cluster.

**Resilience** The security solution should be resilient to the failure of security servers. Either the security service should be highly available or security should be possible if the service is unavailable.

**Dynamic** The security solution should be able handle rapid changes in cluster membership.

**Untrusted nodes** The security solution should mitigate against the threat of cluster nodes acting to disrupt the computation.

**Performance** The security solution should not significantly impact performance of the MapReduce application.

As we have detailed, symmetric cryptographic methods of peer-entity and message authentication, such as Kerberos, do not scale well. Every pair of entities requires a shared symmetric key for security, resulting in $n^2$ keys required in a system of $n$ entities. This may be mitigated by a trusted third party (TTP) which negotiates ephemeral keys on behalf of the entities. However, scaling the TTP service is problematic and the service must be available for entities to be able to authenticate. Furthermore, as we may be deploying the MapReduce application in an untrusted environment, such as a public cloud, then the key escrow problem becomes significant. If the TTP server becomes compromised, then all keys
used for security are compromised. Finally, if we have an elastic MapReduce application where nodes are rapidly added, then automatically negotiating a client master key between a new node and the TTP securely becomes a major problem. Gong’s secret sharing protocol [151] addresses both scalability and resilience of the TTP service and key escrow. However, it performs poorly, requiring $\Omega(n)$ protocol messages to negotiate a session key, and does not scale dynamically.

As an alternative, we could consider public key cryptography for peer-entity and message authentication and for secure negotiation of symmetric cryptographic keys for storing and transferring data confidentially. An X.509 public key infrastructure (PKI) would seem to scale easily by adding subordinate CAs responsible for issuing and revoking certificates. However, the PKI is not resilient to the failure of a CA. If a CA fails then all certificates it has issued are questionable. Without the publication of new revocation lists, we have no means of determining of the currency of a certificate. In addition, if a client has not already retrieved the CAs certificate, it will be unable to validate any certificates the CA has issued. Also, public key methods do not scale dynamically. If we introduce a new node, then its public key must be certified. This process must be adequately authenticated and secure to avoid an impersonator submitting keys to be certified. As a result, we would require another authentication system for certificate requests. The latter issue could be addressed by identity-based or certificateless public key cryptography. However, these systems are dependent on the availability of the private key generator. Another alternative is Lin et al.’s [208] secret sharing PKI. This scheme is scalable and resilient. However, this system is vulnerable to attack when the new PKI server is elected as outlined in section 3.3.3.

Similarly, the papers in section 3.3.3 detail many schemes that add authorisation and encryption to the MapReduce distributed file system (DFS). However, if the DFS cluster is insecure, such as in a cloud, then compromising file storage nodes will allow an attacker to gain access to cryptographic keys used to encrypt files and to bypass authorisation policies. Cohen et al. [91] describe a system where the hardware trusted platform module (TPM) is used to secure cryptographic keys. Whether we use hardware supported encryption or not, if an attacker gains control of the hardware, these schemes become redundant. For a file to be used it must be unencrypted. Therefore, at some point the unencrypted file will be resident in RAM. For most machines, RAM is insecure, and this represents a significant
side channel for an attacker to gain access to unencrypted data. Indeed, using such a side channel, cold-boot attacks have been shown to be effective against TPM-based file encryption [154].

On a related note, section 3.3.3 also discusses the use of secure hardware to support MapReduce computation [295, 318]. In this case, the security of the computation is based on the security of trusted hardware. The interface between the secure and insecure hardware represents a potential risk. Davenport and Ford [104] discuss such risks regarding Intel’s SGX [11] technology. They conclude that while SGX offers tremendous benefits, it does not offer complete security. Indeed, they discuss potential attacks on the platform.

However, sophisticated authentication, authorisation, and confidentiality methods will not prevent a “rogue” system administrator or a privileged process on a compromised cluster node from accessing confidential data. Such an entity has the authorisation to access private keys and confidential data, to alter the user-defined MapReduce programs, or any other malicious activity against the MapReduce application.

Therefore, we propose that a MapReduce system that computes on encrypted data is the only solution that will negate the threat of a privileged insider, such as a cloud provider system administrator. In such a system, the data would be encrypted in a trusted location and then loaded into the insecure cloud MapReduce platform. Such a system would keep the data private from “snoopers”, as the data would remain encrypted on the computation nodes throughout the computation. Furthermore, should a node become compromised, we have not exposed any private cryptographic keys as they are not stored on the nodes. We should note that such a solution does not prevent “vandalism”, where an attacker simply wishes to disrupt the computation, particularly where computed data is altered. However, if we combine our proposal with a result verification method, we can mitigate against such attacks.

4.6 Chapter Summary

This chapter provided a detailed analysis of MapReduce job execution. We examined the flow of requests and responses from job initiation to job completion.
This job execution flow informed a threat analysis which identified the key threats to the MapReduce application as a result of these requests and responses. Finally, we discussed these threats and offered some insight into how these threats may be countered. As a result of this analysis and the literature review in the previous chapter, we have identified private MapReduce computation as our solution for secure MapReduce computation. Section 4.5 provides our justification for why this is our chosen method.
Chapter 5

Cryptographic Security Models

5.1 Introduction

In the last chapter, we concluded that computing over encrypted data is a promising approach for a secure computation in the cloud system. Therefore, in this chapter, we now provide formal definitions of symmetric encryption and message authentication codes, which we will use in the following chapters. We also review cryptographic attack models and we provide definitions for four semantic security games based on these attack models. These security games are the basis of commonly used security definitions in the literature. These definitions will inform the details and discussions of chapters 6 to 9.

5.2 Symmetric Encryption and Message Authentication Codes

In this section, we formally define what we mean by a symmetric encryption scheme and a message authentication code scheme. Informal definitions for both were give in section 3.1.2.

A symmetric encryption scheme is a set of three algorithms: KGen, a key generation algorithm; Enc, the encryption algorithm; and Dec, the decryption algorithm.
5.3. ATTACK MODELS

**Definition 5** (Symmetric encryption scheme.). A symmetric encryption scheme, $\mathcal{SE}$, is the system of algorithms $(\text{KGen}, \text{Enc}, \text{Dec})$. We denote $k \leftarrow KGen(\lambda)$ for the process of (pseudorandomly) generating a secret key $k$ given a security parameter $\lambda \in \mathcal{S}$. For a plaintext $m$, we denote $c \leftarrow \text{Enc}(k, m)$ as the encryption of $m$. We denote $\text{Dec}(k, c)$ as the decryption of $c$. $\mathcal{SE}$ is consistent if $m = \text{Dec}(k, c)$.

A message authentication code scheme is a set of three algorithms: $\text{KGen}$, a key generation algorithm; $\text{MAC}$, the message authentication code (MAC) generation algorithm; and $\text{Vf}$, the MAC verification algorithm.

**Definition 6** (Message authentication code scheme.). A message authentication code scheme, $\mathcal{MAC}$, is the system of algorithms $(\text{KGen}, \text{MAC}, \text{Vf})$. We denote $k \leftarrow KGen(\lambda)$ for the process of (pseudorandomly) generating a secret key $k$ given a security parameter $\lambda$. For a plaintext $m$, we denote $t \leftarrow \text{MAC}(k, m)$ as the message authentication code (MAC) or tag of $m$. $\mathcal{MAC}$ is consistent if $\text{Vf}(k, m, t) = \text{True}$

### 5.3 Attack Models

In this section, we describe the models most commonly used to reason about cryptographic systems.

**Definition 7.** Polynomial time or space. Polynomial time or space will mean that the computational time or space is bounded by a polynomial in $\lambda$, the security parameter, i.e. $\text{poly}(\lambda)$.

#### 5.3.1 Ciphertext only attack (COA)

A ciphertext only attack (COA) is where an attacker only has access to a set of ciphertexts generated by a particular instance of an encryption algorithm. The attacker is able to perform any number of polynomial time operations on these ciphertexts in an attempt to deduce the decryption algorithm.
5.3.2 Known plaintext attack (KPA)

A known plaintext attack (KPA) is where, in addition to having a set of cipher-
texts, an attacker also has a smaller set of pairs of plaintexts and their correspond-
ing ciphertexts. Again, using this data, an attacker is able to perform operations in an attempt to break the system.

5.3.3 Chosen plaintext attack (CPA)

A chosen plaintext attack (CPA) is where an attacker is able to have a lim-
ited number of plaintexts encrypted on their behalf. The black-box encrypter only returns the ciphertexts generated from each submitted plaintext. Using the ciphertexts returned along with other visible ciphertexts, the attacker attempts to deduce the decryption operation in polynomial time.

5.3.4 Chosen ciphertext attack (CCA)

A chosen ciphertext attack (CCA) is similar to CPA. However, in this case, the attacker is also able to submit ciphertexts to a black-box decrypter which returns the plaintexts to the attacker.

5.4 Provable Security

For many cryptographic systems, such as block ciphers, their security is unknown. Empirically, we can show that they are resistant to statistical attacks by subjecting them to tests designed to expose flaws in a cryptographic system. In addition, the complexity of the encryption algorithm may make it unclear as to how one would compute the inverse function without knowledge of secret parameters.

However, there are some cryptographic systems for which we can prove properties of the system which provide a notion of the security of the algorithms. Shannon produced the first proof of security for a cryptosystem when he showed that a one-time pad satisfies perfect secrecy. Perfect secrecy is a special case of information-theoretic security, where an adversary simply does not have
5.4. PROVABLE SECURITY

enough information about the cryptographic system to break it even with unlimited computational power. For perfect secrecy, an adversary cannot deduce anything about a plaintext from a ciphertext without knowledge of the secret key used to generate the ciphertext. We paraphrase Shannon’s definition \[298\] here:

**Definition 8 (Perfect secrecy).** *By Bayes’ Theorem, we have,*

\[
P_c(m) = \frac{P(m)P_m(c)}{P(c)}
\]

(5.1)

where \(P(m)\) is the a priori probability of message \(m\), \(P_m(c)\) is the conditional probability of ciphertext \(c\) given message \(m\) is chosen, \(P(c)\) is the probability of ciphertext \(c\), and \(P_c(m)\) is the a posteriori probability that ciphertext \(c\) is an encryption of message \(m\).

*Therefore, for perfect secrecy, \(P_m(c) = P(c)\) for all \(m\) and \(c\), i.e. \(P_m(c)\) is independent of \(m\)*

Perfect secrecy is a very strong notion of security. It is invulnerable to brute-force attacks. Trying all possible plaintext values does not yield any information about which value of \(m\) for which \(c\) is an encryption. Furthermore, Shannon showed that any system which satisfies perfect secrecy must be essentially equivalent to a one-time pad, i.e. have a key as least as large in bits as the plaintext and that the key must be secret and not be reused.

However, for most cryptographic systems, information-theoretic and perfect secrecy are too strong as notions of security. Therefore, we have proofs of security arising from one or both of two notions: reduction to a “hard” problem (reductionist security) \[29\] and semantic security \[146\]. Reductionist security arguments rest on the fact that a cryptosystem is constructed from a “primitive” which is a trapdoor function, i.e. a function for which is easy to compute the inverse with knowledge of the secret parameters but computation of the inverse without knowledge of the secret parameters is hard. Using a reductionist argument, a polynomial time algorithm can be constructed which can transform an instance of a problem which is computationally hard, or believed to be so, to the problem of computing the inverse of the one-way function without knowledge of the secret parameters. Similar reductionist arguments are used in computational complexity theory to show that a problem exists in a particular complexity class.
For example, demonstrating that the 3-satisfiability (3SAT) problem can be reduced to the problem in question proves that the new problem is NP-complete because 3SAT is known to belong to the NP-complete class. Rabin produced the first system that was provably equivalent to prime factorisation \cite{276}. Whether factorisation is NP-hard is not known, but the problem is believed to hard, and no polynomial time algorithm is known for factorisation.

Semantic security is a probabilistic notion of security. Semantic security was first introduced in 1982 by Goldwasser and Micali in \cite{146}. Informally, semantic security is achieved if a polynomial-time adversarial algorithm, that is given a ciphertext $c$ and the length of the message $m$ from which $c$ was generated, cannot determine any information about $m$ that any polynomial-time algorithm which only has the message length could derive. In \cite{147}, Goldwasser and Micali proved that semantic security was equivalent to the ciphertext indistinguishability under CPA property. Following this model, other similar properties, such as ciphertext indistinguishability under CCA \cite{32,33,35}, have been defined for public-key and symmetric encryption. We discuss these notions below.

A further development is concrete security or practice-oriented provable security \cite{28,29}. This attempts to provide a quantitative analysis of the security of a system. The proofs are formulated similarly to semantic security games (see section 5.5). However, this is expressed in terms of numerical parameters. These parameters can then be set to values which make an attacker’s chances of determining a plaintext no better than guessing. These concrete definitions rely on an assumption that there is no algorithm for the underlying problem beyond what is currently known.

### 5.5 Semantic Security Games

In this section we define the commonly used semantic security properties for symmetric encryption. We formally define three indistinguishability properties commonly referenced in the literature: ciphertext indistinguishability under CPA (IND-CPA), ciphertext indistinguishability under non-adaptive CCA (IND-CCA1), and ciphertext indistinguishability under adaptive CCA (IND-CCA2). As the work of this thesis is concerned with symmetric encryption only, we will
provide definitions for that focus. However, similar definitions exist for public-key encryption.

For each definition, we describe the security game and then provide formal definitions for the actors involved. These definitions are derived from [28, 34, 35, 147] but are reformulated to make the actions of the adversary algorithm explicit as well as make the security game clearer to understand.

5.5.1 Ciphertext indistinguishability under CPA (IND-CPA) for symmetric encryption schemes

The security game for IND-CPA is described as follows

1. A challenger initiates the encryption algorithm $\text{Enc}$ using a secret key $k$.

2. The adversary makes computations on its ciphertext and plaintext inputs and submits plaintexts to the black-box encrypter for encryption.

3. The adversary supplies equal length inputs $m_0, m_1$.

4. The challenger randomly chooses integer $b$ in $[0,1]$.

5. The challenger returns $c_b = \text{Enc}(m_b, k)$ to the adversary.

6. The adversary is able to make additional computations or submit plaintexts for encryption.

7. The adversary returns a value $d$.

8. If $b = d$, the adversary wins.

Given this game, we define IND-CPA as the property that the adversary cannot determine $b$ given $c$ in polynomial time with probability one-half plus $\text{negl}(\lambda)$, i.e. it cannot distinguish one ciphertext from another.

We now formalize the security game given above. We first define the challenger, the left-or-right oracle, $\mathcal{LR}$, which also fulfils the role of the black-box encrypter.
Definition 9 (Left-or-right oracle). A left-or-right oracle, \( \mathcal{LR} \), for symmetric encryption scheme \( \mathcal{SE} \), where \( \mathcal{SE} \) is defined as above, is a random function which on equal length inputs \( m_0, m_1 \in \mathcal{M} \) returns \( c_b \leftarrow \text{Enc}(sk, m_b) \) for \( b \xleftarrow{} [0, 1] \) and a secret key \( sk \).

With our definition of the challenger, we can define what the adversary in the security game is able to do:

Definition 10 (CPA adversary). Let \( A_{\text{CPA}} \), a CPA adversary, be an algorithm which is able to make a polynomially-bounded number of polynomial time computations on its plaintext and ciphertext inputs, including submitting pairs of equal length plaintexts to \( \mathcal{LR} \), a left-or-right oracle, for encryption. \( A_{\text{CPA}} \) chooses a particular pair \( (m_0, m_1) \) to submit to \( \mathcal{LR} \) and receives the response \( c_b \). \( A_{\text{CPA}} \) can also make a further polynomially-bounded number of polynomial time computations after receiving the response. \( A_{\text{CPA}} \) outputs a value \( d \in [0, 1] \).

With the previous two definitions we are now able to define what is meant by IND-CPA security.

Definition 11 (Ciphertext indistinguishability under CPA (IND-CPA)). Let \( \mathcal{SE} \) be a symmetric encryption scheme as above. Let \( A \) be a CPA adversary. Then \( \mathcal{SE} \) is IND-CPA secure if for any \( A \), the probability that \( A \) returns \( d \) where \( d = b \), given \( c_b \), is not better than \( \frac{1}{2} + \varepsilon \) where \( \varepsilon = \text{negl}(\lambda) \).

5.5.2 Ciphertext indistinguishability under non-adaptive CCA (IND-CCA1) for symmetric encryption schemes

The security game for IND-CCA1 is described as follows:

1. A challenger initiates the encryption algorithm \( \text{Enc} \) using a secret key \( sk \).
2. The adversary makes computations on its ciphertext and plaintext inputs and submits ciphertexts to a black-box decrypter for decryption.
3. The adversary supplies inputs \( m_0, m_1 \).
4. The challenger randomly chooses integer \( b \) in \([0,1]\).
5. The challenger returns $c_b = \text{Enc}(m_b, sk)$ to the adversary.

6. The adversary is able to make additional computations.

7. The adversary returns a value $d$.

8. If $b = d$, the adversary wins.

The challenger is a left-or-right oracle as defined above. We define the black-box decrypter as a decryption oracle:

**Definition 12** (Decryption oracle). A decryption oracle, $\mathcal{D}$, for symmetric encryption scheme $\mathcal{SE}$, where $\mathcal{SE}$ is defined as above, is a function which on inputs $c$ returns $m \leftarrow \text{Dec}(sk, c)$ for $c \in C$ and secret key $sk$.

We now formally define the adversary’s capabilities:

**Definition 13** (Non-adaptive CCA adversary). Let $A_{\text{CCA1}}$, a non-adaptive CCA adversary, be an algorithm which is able to make a polynomially-bounded number of polynomial time computations on its plaintext and ciphertext inputs, including submitting ciphertexts to $\mathcal{D}$, a decryption oracle, for decryption. $A_{\text{CCA1}}$ chooses a particular pair $(m_0, m_1)$ to submit to $LR$ and receives the response $c_b$. $A_{\text{CCA1}}$ can also make a further polynomially-bounded number of polynomial time computations after receiving the response but cannot submit further ciphertexts to the decryption oracle $\mathcal{D}$. $A_{\text{CCA1}}$ outputs a value $d \in [0, 1]$.

Therefore our definition is IND-CCA1 security is:

**Definition 14** (Ciphertext indistinguishability under non-adaptive CCA (IND-CCA1)). Let $\mathcal{SE}$ be a symmetric encryption scheme as above. Let $A$ be a non-adaptive CCA adversary. Then $\mathcal{SE}$ is IND-CCA1 secure if, for any $A$, the probability that $A$ returns $d$ such that $d = b$, given $c_b$, is not better than $1/2 + \varepsilon$ where $\varepsilon = \text{negl}(\lambda)$.

### 5.5.3 Ciphertext indistinguishability under adaptive CCA (IND-CCA2) for symmetric encryption schemes

The security game for IND-CCA2 is described as follows:
1. A challenger initiates the encryption algorithm $\text{Enc}$ using a secret key $sk$.

2. The adversary makes computations on its ciphertext and plaintext inputs and submits ciphertexts to the black-box decrypter for decryption.

3. The adversary supplies inputs $m_0, m_1$.

4. The challenger randomly chooses integer $b$ in $[0,1]$.

5. The challenger returns $c_b = \text{Enc}(m_b, sk)$ to the adversary.

6. The adversary is able to make additional computations and submit ciphertexts to the black-box decrypter for decryption provided the ciphertext is not $c_b$.

7. The adversary returns a value $d$.

8. If $b = d$, the adversary wins.

Given the definitions for the left-or-right oracle and decryption oracle above, we define the capabilities of the adversary.

**Definition 15** (Adaptive CCA adversary). Let $A_{CCA2}$, an adaptive CCA adversary, be an algorithm which is able to make a polynomially-bounded number of polynomial time computations on its plaintext and ciphertext inputs, including submitting ciphertexts to $D$, a decryption oracle, for decryption. $A_{CCA2}$ chooses a particular pair $(m_0, m_1)$ to submit to $LR$ and receives the response $c_b$. $A_{CCA2}$ can also make a further polynomially-bounded number of polynomial time computations after receiving the response, including submitting ciphertexts to $D$ for decryption, but cannot submit $c_b$ to $D$. $A_{CCA2}$ outputs a value $d \in [0,1]$.

Now we define what we mean by IND-CCA2 security.

**Definition 16** (Ciphertext indistinguishability under adaptive CCA (IND-CCA2)). Let $SE$ be a symmetric encryption scheme as above. Let $A$ be an adaptive CCA adversary for $SE$. Then $SE$ is IND-CCA2 secure if, for any $A$, the probability that $A$ returns $d$ such that $d = b$, given $c_b$, is not better than $\frac{1}{2} + \varepsilon$ where $\varepsilon = \text{negl}(\lambda)$. 
5.5. SEMANTIC SECURITY GAMES

5.5.4 Unforgeability under Chosen Message Attack (UF-CMA)

In chapter 9 we construct a searchable symmetric encryption scheme using a message authentication code scheme as one its primitive building blocks. Therefore, in this section, we present a security game for UF-CMA and a set of formal definitions.

The security game is:

1. A challenger initiates the MAC signing algorithm MAC using a secret key sk. The size of the tag space, \( T \), is superpolynomial in the security parameter \( \lambda \).

2. The adversary makes computations on its plaintext and tag inputs and submits plaintexts to a black-box signer for signing.

3. The adversary supplies a pair \((m, t)\) to the challenger where \( t \) has not previously been submitted to the black-box signer.

4. The challenger returns \( b = \text{Vf}(m, t, sk) \) to the adversary.

5. If \( b = \text{True} \), the adversary wins.

To formalise this definition we first define what we mean by the challenger, a signing oracle. The signing oracle also fills the role of the black-box signer.

Definition 17 (Signing oracle). A signing oracle, \( \mathcal{T}O \), for the message authentication code scheme \( \text{MAC} \), as defined above, is a function which on input \( m \) returns \( \text{MAC}(m, sk) \) for secret key \( sk \).

We can now define the capabilities of the adversary:

Definition 18 (Chosen Message Attack (CMA) adversary). Let \( A_{CMA} \), a CMA adversary for message authentication code scheme \( \text{MAC} \), where \( \text{MAC} \) is defined as above, be an algorithm which is able to make a polynomially-bounded number of polynomial time computations on its plaintext and tag inputs, including submitting plaintexts to \( \mathcal{T}O \), a signing oracle, for signing. Let \( \mathcal{M}^* \) be the set of plaintexts submitted to \( \mathcal{T}O \). \( A_{CMA} \) chooses a plaintext \( m \notin \mathcal{M}^* \) and outputs the pair \((m, t)\) where \( t \) is \( A_{CMA} \)'s computed value for the tag of \( m \).
Therefore, we define UF-CMA security as:

**Definition 19** (Unforgeability under Chosen Message Attack (UF-CMA)). Let $\text{MAC}$ be a message authentication code scheme. Let $A$ be a CMA adversary for $\text{MAC}$. Then $\text{MAC}$ is UF-CMA secure if, for any $A$, the probability that $A$ returns $(m, t)$ such that $V_f(m, t, sk) = True$ is not better than $1/|\mathcal{T}| + \varepsilon$ and $\varepsilon = \text{negl}(\lambda)$.

### 5.6 Criticisms

In this section, we discuss criticisms of the provable security methodology commonly used in analysing cryptosystems. An extensive critique of provable security is given in [194, 195], particularly with regard to reductionist arguments. We add further criticisms of our own.

Firstly, it should be noted that a proof of security is not necessarily a proof that a system is unbreakable, or unforgeable in the case of MACs. Reductionist proofs rely on a “hardness” assumption. However, showing that a system is equivalent to a problem in a particular complexity class may not be sufficient.

Consider the knapsack problem [101, 217], for which the decision problem is NP-complete. However, this result was slightly surprising as it ran contrary to a general belief that the problem was computationally easy. The reason for this was that most instances studied had been easy to compute. Several cryptosystems have been based on the knapsack problem, such as the Merkle-Hellman system [222]. However, most have been shown to be breakable [252]. The reason for this is that, for cryptographic purposes, we must find an instance of the knapsack problem which is easy to solve with knowledge of the secret parameters but which is hard to solve without knowledge of the secret parameters. However, finding such instances has proven challenging. Therefore, in the case of schemes such as Merkle-Hellman, a reductionist proof of security was not sufficient to prove that the system was unbreakable.

Similarly, some semantic or concrete security proofs do not prove the system is unbreakable. As we discuss in section 8.2.2, Xiao and Yen [353] prove that an order-preserving encryption scheme for the domain $[1, 2]$ is IND-OCPA secure [53]. However, as result of the order-preserving property, a system for such a
5.6. CRITICISMS

small domain is trivially breakable.

With regard to reductionist proofs, we should note that the argument relies on the assertion that no polynomial time algorithm can be found which can solve the “hard” problem. However, this does not consider that a superpolynomial algorithm may be sufficient to solve the problem in a reasonable time for typical instances of the problem.

Semantic security proofs (and concrete security proofs constructed in a similar fashion) of IND-CPA, IND-CCA1 and IND-CCA2 rely on the ability of an adversary to encrypt messages and decrypt ciphertexts. For public-key encryption, an adversary may encrypt any message and inspect the ciphertext because the encryption algorithm and key are made public. Therefore, IND-CPA seems like an eminently sensible security model as it corresponds to real-world practice. However, for the CCA models and for symmetric encryption, the assumption that an adversary has access to pairs of corresponding plaintext and ciphertext pairs is dependent on the application and usage scenario. The scenario used in [195] to justify the CCA models is unconvincing because it goes against common industry practice along with common sense. Koblitz and Menezes [195] posit a scenario where a system has crashed and an attacker poses as a legitimate user trying to recover data. In such a case, users would not simply have data decrypted for them on demand by the system administrators. It should be pointed out that, in some systems, the decryption key is only possessed by the legitimate user, making this attack pointless. Even supposing the administrator could decrypt all files, they would not do so for an unauthenticated request as described in this scenario. Rather, the attacker would be advised to decrypt the data using their previously stored key. Similarly, with regard to symmetric-key encryption, the scenarios used to justify the models for symmetric-key schemes in [34, 54] are similarly unconvincing as they place unrealistic restrictions on an attacker’s actions. Bellare and Rogaway [34] describe a “lunchtime attack” where a workstation is left accessible to an attacker for a short period. The attacker is able to generate or decrypt ciphertexts as required. However, in this scenario an attacker could easily install a keylogger to capture plaintexts as they are typed, thereby circumventing the encryption. Boneh and Shoup [54] suggest a scenario where emails received into a mailbox are automatically encrypted. An attacker sends an email to a recipient and then, having broken into the system, is able to inspect the
encrypted email. However, in this scenario it would be far more effective for the attacker to search for the stored encryption key, thereby allowing one to decrypt all the email.

It is typical in the literature to suppose that a proof of IND-CPA, IND-CCA1, etc. security is a necessary proof of security. However, does such a security model adequately capture the practical security requirements of the system? In practice, an adversary will have to conduct ciphertext-only attacks on a symmetric-key scheme. Similarly, the chances of a known plaintext attack are unlikely if industry best practices are followed. It is well understood that data that is typically replicated such as protocol headers should not be encrypted to avoid KPA. We should note the concrete security attempts to address the practical security requirements of a cryptosystem by specifying bounds on the parameters of the system, such as the key bit length. However, these proofs are typically framed in terms of the security models described above. Therefore, the question of whether the model adequately captures the requirements still persists.

The ubiquity of the security game model of security has led to definitions, such as IND-OCPA \cite{53}, which do not seem practically realistic. Proving that a system satisfies such a definition says nothing about its practical security. It should also be noted that security game setup of semantic security proofs would appear to imply that such proofs are independent of a “hardness” assumption. This may lead one to misapply such techniques to falsely claim that a system is secure.

We finally note that the security of the system may depend more on its implementation than its provable security properties, as Koblitz points out \cite[p2]{195} with regard to Bleichenbacher’s attack on RSA \cite{46}. Bleichenbacher’s attack on RSA exploited the protocol in the implementation regarding formatting of the plaintext before encryption. So while a cryptosystem may satisfy a provable security definition, the practical implementation may be insecure.

In closing, we acknowledge that security proofs are useful guides to the practical security of a cryptosystem. However, we also note that such techniques should be carefully applied and that security models should try to adequately capture the practical security requirements.
5.7 Conclusion

In this chapter, we have provided formal definitions for the security models which will be discussed in later chapters. We have also critically evaluated their use in practice. This evaluation informs the following chapters, particularly with regard to which attack models we feel are appropriate to the usage scenarios described in chapters 6 to 9.
Chapter 6

CryptMR: Secure Computation in the Cloud Using MapReduce

6.1 Introduction

As concluded in chapter 4, being able to perform MapReduce (MR) computations over encrypted data is how we intend to approach the problem of secure MR computation in the cloud. In this chapter, we provide a formal model of our expected usage scenario. We then provide a high-level overview of our solution (CryptMR) including the encryption schemes we will require for our solution. The concrete details of these schemes will be discussed in chapters 7 to 9.

6.1.1 Challenges for Secure MR in the Cloud

As we discussed in chapters 2 and 4, we require a means to process data securely when that computation is outsourced to the cloud. We designate such a computation, *secure computation in the cloud* (SCC). SCC should not expose input or output data to any other party, including the cloud service provider. Furthermore, the details of the computation should not allow any other party to deduce its inputs and outputs. With regard to MR, we have three sets of data that must be protected. The first is the input data which is stored in the distributed file system (DFS) and which is processed by the mappers. The second set is the intermediate data generated by the mappers as a result of processing the user-defined
map function on the input data. This data is stored locally on each mapper. The last set is the output data generated by the reducers as a result of applying the user-defined reduce function on the intermediate data. The output data is stored in the DFS. These data sets are at risk of exposure when at rest or when transmitted between DFS or computation nodes. In the case of transmission, the communication can be protected by a well understood secure protocol such as SSL. Indeed, Hadoop implements SSL communication between nodes [102, 251]. In the case of data at rest, several solutions have detailed encrypting the DFS [91, 94, 108, 175, 207, 215, 259, 266]. While these approaches protect the data in the DFS, when it is consumed by the MR cluster nodes it must be unencrypted in order to be processed. Ideally, we should like to process the data in its encrypted form so that at no point in the computation process it is exposed in an unencrypted form. More challengingly, this data must be encrypted so that the encrypted results of the computation are meaningful once decrypted. Cryptography seems the natural approach to this problem.

However, it should be noted that van Dijk and Juels [112] show that cryptography alone cannot realise secure multi-party computation in the cloud, where the parties jointly compute a function over their inputs while keeping their own inputs private. Since our approach is via homomorphic encryption, we will restrict our attention to what we will call secure single-party computation in the cloud (SSCC).

Homomorphic encryption (HE) seems to offer a solution to the SSCC problem. However, despite the clear advantages of FHE, and many significant advances [65, 67], it remains largely impractical, as discussed in [3.2.1]. Therefore, we take the view in our work that only SHE is, for the foreseeable future, of practical interest. Our goal is to develop new SHE schemes which are practically useful, and which we have tested with a realistic implementation.
6.2 Background

6.2.1 Scenario

As introduced above, our work concerns secure single-party computation in the cloud. In our scenario, a secure client wishes to compute a function on a large volume of data. This function could be searching or sorting the data, computing an arithmetic function of numeric data, or any other operation. We consider here the case where the client wishes to perform arithmetic computations on numeric data. This data might be the numeric fields within a record, with non-numeric fields being treated differently.

The client delegates the computation to the cloud. However, while the data is in the cloud, it could be subject to snooping, including by the cloud provider. The client does not wish to expose the input data, or the output of the computation, to possible snooping in the cloud. A snooper here will be a party who may observe the data and the computation in the cloud, but cannot, or does not, change the data or insert spurious data. (In our setting data modification would amount to pointless vandalism.) The snooping may be casual, displaying an uninvited interest, or malicious, intending to use data for the attacker’s own purposes.

To obtain the required data privacy, the client’s function will be computed homomorphically on an encryption of the data. The client encrypts the source data using a secret key. This source data set is static and is not updated during the computation. The client then uploads the encryptions to the cloud, with a homomorphic equivalent of the target computation. It is not The cloud environment performs the homomorphic computation on the encrypted data. The result of the homomorphic computation is returned to the client, who decrypts it using the secret key, and obtains the output of the computation.

In this scenario, the source data is never exposed in the cloud, but encryptions of it are. A snooper may observe the computation of the equivalent homomorphic function in the cloud environment. As a result, they may be able to deduce what operations are performed, even though they do not know the inputs. A snooper may also be able to inspect the (encrypted) working data generated by the cloud computation, and even perform side computations of their own. However, snoopers have no access to the secret key, so cannot make encryptions
of their own.

Our scenario does not address preventing or mitigating side-channel attacks. Such attacks are outside the scope of our work. While a privileged insider would be able to perform such attacks (see the discussion of side-channel attacks in sections 4.3.7 and 4.5) we make the assumption that they do not have the motivation to go to such lengths.

6.2.2 Formal Model of Scenario

We have $n$ inputs $m_1, m_2, \ldots, m_n$ distributed in $\mathcal{M}$ according to a probability distribution $\mathcal{D}$. We wish to compute a function $f$ on these inputs. A secure client $A$ selects the secret parameter set $K$. $A$ encrypts the $n$ inputs by computing $c_i \leftarrow \text{Enc}(K, m_i)$, for $i \in [1, n]$. $A$ uploads $c_1, c_2, \ldots, c_n$ and $P'$ to the cloud computing environment, where $f'$ is the homomorphic equivalent of $f$ in the ciphertext space. These encryptions do not all need to be uploaded at the same time but $n$ is a bound on the total number of inputs. The cloud environment computes $f'(c_1, c_2, \ldots, c_n)$. $A$ retrieves $f'(c_1, c_2, \ldots, c_n)$ from the cloud, and computes

$$f(m_1, m_2, \ldots, m_n) = \text{Dec}(K, (f'(c_1, c_2, \ldots, c_n))).$$

A snooper is only able to inspect $c_1, c_2, \ldots, c_n$, the function $f'$, and the computation of $f'(c_1, c_2, \ldots, c_n)$, including subcomputations and working data, and perform side-computations on these. Thus the snooper is passive or honest-but-curious.

6.2.3 Observations from Scenario

From our scenario we observe that we do not require public-key encryption as we do not intend another party to encrypt data. Symmetric encryption will suffice. Furthermore, there is no key escrow or distribution problem, as only ciphertexts are distributed to the cloud.

Note that the $n$ inputs do not necessarily need to be uploaded at once, but $n$ is
an upper bound on the total number of inputs. For example, if the function $f$ can be decomposed, we might compute it in separate stages, and this might be useful in more dynamic situations.

This model is clearly susceptible to certain attacks. We consider ciphertext only, brute force, and cryptanalytic attacks. To avoid cryptanalytic attacks, we must choose the parameters of the cryptosystem carefully. Here, a brute force attack will mean guessing the plaintext associated with a ciphertext. In our model, known plaintext attack (KPA) is considered possible only by brute force, and not through being given a sample of pairs of plaintext and corresponding ciphertext.

We observe that, since the source data set is static, an attacker is not able to interactively submit plaintexts to the data owner to be encrypted so that he can then view their encryptions in the cloud. Furthermore, even if an attacker submits a plaintext to the data owner for inclusion in the source data set, no data is uploaded to the cloud that is not used in the computation. Furthermore, no data is uploaded unencrypted. This prevents an attacker linking submitted plaintexts to their encryptions in the cloud. Therefore, chosen plaintext attacks are infeasible in this scenario. Similarly, as the data owner is the only party that can view the decrypted data, chosen ciphertext attacks are also infeasible.

We note that observation of the function $f'$, which closely resembles $f$, might leak some information about its inputs. By examining the individual operations we might be able to deduce, for example, whether an input was numerical or textual data, or whether the data is comparable and hence infer the overall nature of the computation. However, we assume that this information is far too weak to threaten the security of the system, as is common in the HE literature. However, if the threat is significant, “garbled circuits” [145] are a possible solution.

Finally, we note that our model of SSCC is very similar to the model of private single-client computing, described in [112]. Furthermore, they describe an example practical application, a privacy preserving tax return preparation program, which computes the relevant statistics on government servers without revealing the client’s inputs. Another example, cited in [199], is a device which collects health data which is streamed to the cloud. Statistics are computed on the data and reported back to the device. To protect the patient’s privacy this data is encrypted by the device and the computations are performed homomorphically.
Erkin et al. [128] employ a similar scenario in the description of their privacy-preserving face recognition algorithm.

### 6.2.4 Assumptions

We assume that the MR DFS and MR framework software can be trusted to perform as expected. However, we do not trust that the DFS or local storage on each of the MR cluster nodes cannot be accessed by a third-party to which we do not wish to disclose the data.

### 6.3 Secure Computation in the Cloud Using MR

In this section we detail our private MR solution. This solution allows for data processing to be performed on encrypted data.

In our solution, the input data is encrypted by a trusted party. This encrypted input data is placed in the MR DFS. The data is processed in encrypted form by the mappers and reducers without being unencrypted. The reducers place the encrypted output data in the DFS. This output data is retrieved by the trusted party and decrypted.

As definitions 1 and 2 in section 2.3 show, the user-defined map and reduce functions in the MR programming model take a key-value pair as an input. However, the datatype of each element in the pair is unspecified. However, this flexibility makes converting the MR programming model to one which operates on encrypted data challenging. One could employ fully homomorphic encryption (section 3.2.1) which allows for arbitrary Boolean circuits to be computed on encrypted data. In this scheme, map and reduce functions would be converted to arithmetic circuits. However, as discussed in section 3.2.1 such schemes are not yet very practical.

Another approach is to inspect the user-defined map and reduce function implementations and convert them to equivalent functions that operate on encrypted data. This has the advantage that one can easily deduce the type information for
data used in computing the map and reduce functions. Furthermore, we can then use this type information to choose an appropriate encryption scheme for each field of data. We can match the encryption scheme to the data type. This allows one to choose the most appropriate and efficient encryption scheme for each field. Furthermore, it makes it easier to generate versions of operations on encrypted data. For example, an addition operation on unencrypted integers, entails an equivalent homomorphic operation on encrypted integer data. In such a scheme, integer data might be encrypted using a homomorphic scheme over the integers. Comparable data might be encrypted using an order-preserving cryptosystem (section 3.2.3). Finally, text data might be encrypted by a scheme that allows for text searching. This ability to choose an appropriate cryptographic scheme for the data offers a significant performance advantage over a more general purpose fully homomorphic encryption scheme. MRCrypt [313] uses such an approach.

We have adopted a similar approach to the above. We have developed a library of homomorphic cryptographic functions to support a range of suitable operations. Furthermore, we have developed an adjunct library that implements the ciphertexts and their associated operations, for example, addition or multiplication, as software that can be easily inserted into a MR application to support secure data processing. The implementation details are presented in Appendix A. Our approach is generic enough that it can be combined with other approaches easily. We could use MRCrypt’s encryption inference engine to automate the process of modifying an unencrypted MR program to one which employs encryption. Our functions could be ported into Crypsis’ variation of the Apache Pig engine [311]. We should also note that our approach is easily extensible to other spheres of distributed and non-distributed computing.

In our solution, we present several novel encryption schemes. The first is a family of “somewhat” homomorphic schemes for integers that allows for arbitrary degree polynomials to be computed. This scheme supports addition and multiplication operations on the same data to be computed without the need for the re-encrypt operation used in Crypsis. This scheme can be transformed into a fully homomorphic scheme. An FHE scheme is not addressed in similar approaches [272, 311, 313]. This family of schemes is presented and evaluated in chapter 7. The second is an order-preserving encryption (OPE) scheme based on the general approximate common divisor problem [164]. This scheme supports sorting of data
and is, as far as we know, the first OPE scheme to be based on a computation hardness primitive. It is presented and discussed in chapter 8. The third is a symmetric searchable encryption (SSE) scheme. This scheme supports searching of data. It is an extension of the Song et al. scheme \cite{309} to address some of its impracticalities. It is discussed in chapter 9.

### 6.3.1 Required Encryption Schemes

As discussed above, we use different encryption methods to encrypt different data items based on the type of the data and the operations we wish to perform on the data. This section outlines the data encryption schemes that we require for our solution.

**Pseudorandom Encryption (RND)**

The RND scheme provides computational indistinguishability under chosen plaintext attack (IND-CPA). Data encrypted by RND is indistinguishable from random bits under IND-CPA. RND is the most secure encryption scheme in the suite of schemes that we would require. However, no operations could be performed on data encrypted by this method, so we would employ it for data that is not used in the computation. We construct RND using a block cipher, such as AES \cite{238} in a suitable mode of operation, such as cipher block chaining (CBC) \cite{125}, with a random initialisation vector (IV) \cite{288}.

**Deterministic Encryption (DET)**

The DET scheme is one where each application of DET on a plaintext will always yield the same ciphertext. Under this scheme, it is possible to determine equality of plaintexts by testing for equality of their respective ciphertexts. This method could also be implemented using a block cipher. We construct DET using a block cipher, such as AES \cite{238} in a suitable mode of operation, such as cipher block chaining (CBC) \cite{125}, with a constant initialisation vector (IV).
Order Preserving Encryption (OPE)

OPE is an order-preserving encryption scheme (see section 3.2.3). Data encrypted by OPE preserves any ordering on the plaintexts. OPE allows us to compare and order ciphertexts so that when decrypted the plaintexts will be ordered. Our OPE scheme is presented in chapter 8.

Symmetric Searchable Encryption (SSE)

SSE allows for specific strings to be matched to instances in an encrypted field without decrypting the field. In particular, it allows for searching over encrypted text fields. SSE is constructed using the scheme presented in chapter 9.

Homomorphic Encryption (HOM)

HOM allows for functions to be computed on encrypted integer data without decrypting the data that would allow us to compute arbitrary degree polynomials on integer values. As defined in section 3.2.1 in this scheme, a mapping exists from operations on plaintext values to an equivalent operation on ciphertext values and vice versa. Therefore, performing operations on the ciphertexts generated from encrypting a set of plaintexts will result in a ciphertext that decrypts to a plaintext that is the result of equivalent plaintext operations on our original set of plaintexts. HOM is constructed using the scheme presented in chapter 7.

6.3.2 Other Encryption Schemes

Fully Homomorphic Encryption (FHE)

An FHE scheme allows us to compute arbitrary depth Boolean circuits. Therefore, an FHE scheme allows one to compute any function which takes a bit sequence as input. The scheme detailed in chapter 7 can be extended to an FHE scheme (see section 7.6). However, as this scheme is largely impractical, we have not implemented it.
6.4 Conclusion

In this chapter we have presented an outline of a system for computing MR jobs on encrypted data (CryptMR). We have discussed our solution and how it differs from related work. The following chapters, 7 to 9, present the concrete details of our solution.

While our proposed solution is inspired by previous work [272, 313], we believe that by basing our work around novel encryption schemes, it allows us to improve on previous work. Our schemes are devised to significantly improve performance for arithmetic operations (HOM) and sorting (OPE) or to add operations not found in other related work (SSE) [311, 313].
Chapter 7

Homomorphic Encryption Over the Integers for CryptMR

An abbreviated version of this material in this chapter is to be presented at the 16th IMA International Conference on Cryptography and Coding [121].

7.1 Introduction

In this chapter, we present four novel SHE schemes for encryption of integers that are additively and multiplicatively homomorphic. These schemes implement the HOM scheme described in the previous chapter [6] and are capable of computing arbitrary degree polynomials. In this section, we present a summary of our results and a discussion of related work. We present our initial homomorphic scheme in section 7.2 in two variants, HE1 and HE1N. HE1 (section 7.2.1) provides strong security for integers distributed with sufficient entropy. This security derives from the assumed hardness of the partial approximate common divisor problem (PACDP). HE1N (section 7.2.2) guarantees strong security for integers not distributed with sufficient entropy or where the distribution is not known, by adding an additional “noise” term. In addition to the hardness assumption, we prove that HE1N is IND-CPA secure [28]. Section 7.3 describes a further two variants, HE2 and HE2N, which increase the entropy of the plaintext by adding a dimension to the ciphertexts, which are 2-vectors. This further increases the security of these schemes by effectively doubling the entropy. HE2 (section 7.3.1) deals
with integers of sufficient entropy, HE2N (section 7.3.2) with integers without the required entropy or of unknown distribution. HE2N also satisfies IND-CPA. We describe this in some detail, since it appears to be practically useful, and is the simplest version of our general scheme. In section 7.4 we generalise HE2 and HE2N from 2-vectors to $k$-vectors, for arbitrary $k$, in the scheme HE$k$, with noisy variant HE$k$N. These schemes may also be practical for small enough $k$.

We have performed extensive experimental evaluation of the first four schemes presented in this chapter. We report on this in section 7.7. Our results are extremely favourable when compared with other methods. In some cases, our algorithms outperform the running times of directly comparable schemes by a factor of up to 1000, and considerably more than that for fully homomorphic schemes, used in the same context. Finally, in section 7.8 we conclude the chapter.

7.1.1 Formal Model of Scenario

To the formal model of section 6.2.2, we add the following further specifications. Our $n$ inputs $m_1, m_2, \ldots, m_n$ are integers distributed in $[0, M)$ according to the probability distribution $D$. If $X$ is a random integer sampled from $D$, let $\Pr[X = i] = \xi_i$, for $m \in M$. We will consider three measures of the entropy of $X$, measured in bits:

- Shannon: $H_1(X) = - \sum_{i=0}^{M-1} \xi_i \lg \xi_i$.
- Collision: $H_2(X) = - \lg \left( \sum_{i=0}^{M-1} \xi_i^2 \right)$.
- Min: $H_\infty(X) = - \lg \left( \max_{i=0}^{M-1} \xi_i \right)$.

It is known that $H_1(X) \geq H_2(X) \geq H_\infty(X)$, with equality if and only if $X$ has the uniform distribution on $[0, M)$, in which case all three are $\lg M$. We will denote $H_\infty(X)$ by $\rho$, so it also follows that $H_1(X), H_2(X) \geq \rho$. We use the term “entropy” without qualification to mean min entropy, $H_\infty(X)$. Note that $H_\infty(X) = \rho \geq \lg M$ implies $\xi_i \leq 2^{-\rho}$, $i \in [0, M)$, and that $M \geq 2^\rho$.

The function we wish to compute is a multivariate polynomial $P$ of degree $d$ on
these inputs. The cloud environment computes $P'(c_1, c_2, \ldots, c_n)$, where $P'$ is the homomorphic equivalent of $P$ in the ciphertext space.

### 7.1.2 Further Observations from Scenario

In addition to the observations of section 6.2.3, we also observe that while the public parameters of our schemes are exposed to the cloud, they do not provide an encryption oracle.

We also note that, for our scenario, a brute force attack will mean guessing the plaintext associated with a ciphertext. In our encryption schemes, it will be true that guesses can be verified. Since $\xi_i \leq 2^{-\rho}$ for $i \in [0, M)$, the expected number $\mu$ of guesses before making a correct guess satisfies $\mu \geq 2^\rho$. Massey [216] gave a corresponding result in terms of the Shannon entropy $H_1(X)$.

Similarly, probability of any correct guess in $2^{\rho/2}$ guesses is at most $2^{-\rho/2}$. This bound holds if we need only guess one of $n$ inputs, $m_1, m_2, \ldots, m_n$, even if these inputs are not independent. Therefore, if $\rho$ is large enough, a brute force attack is infeasible. An example of high entropy data is salaries for a large national or multinational business. Low entropy data might include enumerated types, such as gender.

### 7.1.3 Our Schemes

We describe new practical HE schemes for the encryption of integers, to be employed in our CryptMR system. Our work is inspired by CryptDB [272] which encrypts integers using the Paillier cryptosystem [262] which is additively homomorphic. Similar systems ([311, 313]) use ElGamal [127] to support multiplications. The “unpadded” versions of these schemes must be used. These are not IND-CPA secure [147], reducing the advantage of a public-key system. These schemes do not support both addition and multiplication. Computing the inner product function requires re-encrypting the data once the multiplications have been done, so that the additions can be performed. In a SSCC system, this requires shipping the data back to the initiator for re-encryption, a significant

---

1Paillier supports computation of linear functions with known coefficients homomorphically by repeated addition
communication overhead. We aim to support both addition and multiplication without this overhead. It should also be noted that a hybrid scheme of Paillier and ElGamal, for a given modulus, will be limited in the degree of polynomials that can be computed. Should a product or sum exceed the modulus then the result cannot be successfully decrypted.

Our scheme is inspired by the SHE scheme of van Dijk et al. that is used as the basis for a public-key system. As in their system, we add multiples of integers to the plaintext to produce a ciphertext. However, \([111]\) supports only arithmetic mod 2. We generalise their scheme to larger moduli.

We showed above that the input data must have sufficient entropy to negate brute force attacks. If the data lacks sufficient entropy, we will introduce more in two ways. The first adds random “noise” of sufficient entropy to the ciphertext, to “mask” the plaintext. This approach is employed in \([111]\). In our “N” variants below, we add a random multiple (from 0 to \(\kappa\)) of a large integer, \(\kappa\), to the ciphertext, such that \(m_i < \kappa\), for all \(i \in [1,N]\). If the entropy of the original data was \(\rho\), it becomes \(\rho + \lg \kappa\). Therefore, if \(\kappa\) is large enough, our data has sufficient entropy. But there is a downside. If the noise term grows too large, the ciphertext cannot be decrypted successfully. So we are restricted to computing polynomials of bounded degree, but this does not appear to be a practical problem.

The other technique will be to increase the dimension of the ciphertext. We represent the ciphertext as a \(k\)-vector, where each element is a linear function of the plaintext. Addition and multiplication of ciphertexts use linear algebra. The basic case \(k = 1\) is described in section \([7.2.1]\). Then we can increase the entropy by creating a \(k\)-vector ciphertext. Then we must guess \(k\) plaintexts to break the system. Assuming that the inputs \(m_1, m_2, \ldots, m_n\) are chosen independently from \(\mathcal{D}\), and the entropy is \(\rho\), the entropy of a \(k\)-tuple \((m_1, m_2, \ldots, m_k)\) is \(k\rho\). Thus the \(k\)-vectors effectively have entropy \(k\rho\). If \(k\) is chosen large enough, we have sufficient entropy to prevent brute force attack. On the upside, some cryptanalytic attacks for \(k = 1\) do not seem to generalise even to \(k = 2\). The downside is that ciphertexts are \(k\) times larger, and each homomorphic multiplication requires \(\Omega(k^3)\) time and space. For very large \(k\), this probably renders the methods impractical. Therefore, we consider the case \(k = 2\) in section \([7.3]\). The general case is considered in section \([7.4]\).

Our work here supports computing arbitrary degree multivariate polynomials on
integer data. However, we expect that for many practical applications, computing low-degree polynomials will suffice. As argued in \[199\] many applications compute linear or quadratic functions, for example, computing a mean or variance of a data set. We do not expect our work to be used in scientific computing where higher degree polynomials might be computed. In this chapter, we present four variants of our scheme. Two provide strong security under the assumption that the input data has high entropy. The other two provide strong security regardless of this assumption. Section 7.4 generalises these four schemes to dimension \(k\) ciphertexts.

7.1.4 Related Work

A comprehensive survey of partial, somewhat and fully HE schemes is presented in \[1\]. In this section, we discuss those most related to our own work in this chapter. Some related work (\[272, 311, 313\]) has already been discussed in chapter 3. Our scheme is inspired by that of van Dijk et al. \[111\]. In their paper they produce a fully homomorphic scheme over the integers where a simple “somewhat” homomorphic encryption scheme is “bootstrapped” to a fully homomorphic scheme. van Dijk et al. take a simple symmetric scheme where an integer plaintext \(m\) is encrypted as \(c = m + 2r + pq\), where \(p\), the secret key, is an odd \(\eta\)-bit integer from the interval \([2^{\eta-1}, 2^\eta)\), and \(r\) and \(q\) are integers chosen randomly from an interval such that \(2r < p/2\). The ciphertext \(c\) is decrypted by the calculation \((c \mod p) \mod 2\). Our scheme HE1N below (section 7.2.2) may be regarded as a generalisation of theirs to arbitrary prime moduli.

van Dijk et al. transform their symmetric scheme into a public key scheme. A public key \(\langle x_0, x_1, \ldots, x_\tau \rangle\) is constructed where each \(x_i\) is a near multiple of \(p\) of the form \(pq + r'\) where \(q\) and \(r'\) are random integers chosen from a prescribed interval. To encrypt a message a subset \(S\) of \(x_i\) from the public key are chosen and the ciphertext is now calculated as \(c = m + 2r + 2 \sum_{i \in S} x_i \mod x_0\). The ciphertext is decrypted as previously described. We could extend our HE1N schemes here to a public key variant, using a similar device. However, we do not do so, since public key systems appear to have very little application in our model.

van Dijk et al. bootstrap their public key system using Gentry’s method \[137\]
to a fully homomorphic scheme. In this case, the bootstrapping is done by ho-
omorphically making a suitable simulation of division by $p$, thus obtaining an
encryption of $c \mod p$ which can be used to continue the computation. Our FHE
proposal is based on entirely different principles.

Coron et al. [81, 95–97] have produced several refinements of the scheme in [111].
In [97], the authors reduce the size of the public key by using a similar but
alternative encryption scheme. In this scheme, $p$ is a prime in the specified
interval, $x_0$ is an exact multiple of $p$ and the sum term in the ciphertext is
quadratic rather than linear. In [96], they apply the Brakerski et al. [65] modulus
switching technique to their system from [97]. In [81], the authors apply the
Smart and Vercauteren optimisations [304] to their scheme. Finally, in [95], they
apply Brakerski’s scale-invariant technique [64] to their system. Pisa et al. [269]
generalise van Dijk et al.’s scheme from base 2 to base $B$ integers. This scheme is
similar to our HE1 scheme. However, our HE1 is a generalisation of van Dijk et
al.’s SHE scheme to an arbitrary prime base, rather than a generalisation of the
public key scheme. Ramaiah and Kumari [279] produce a variant of van Dijk et
al.’s scheme with significantly smaller key sizes. Chen et al. [76] propose a more
efficient re-encryption scheme that enhances van Dijk et al.’s scheme. Aggarwal
produce a scheme for non binary messages. Most recently, Wang et al. [332] have
produce a variant similar to Coron et al.’s original scheme, which reduces the
public key size by making the sum term cubic.

Several implementations of SHE and FHE schemes have been produced. Lauter
et al. [199] implement the SHE scheme from [67]. However, they give results only
for degree two polynomials. Our schemes are capable of computing degree three
and four polynomials for practical key and ciphertext sizes. HELib [156] is an
implementation of the BGV [65] FHE scheme. HELib-MP [283] is an adaptation
of HELib to support multi-precision moduli. At the current time, it only supports
basic SHE features. The Homomorphic Encryption Applications and Technology
(HEAT) project’s Homomorphic Encryption Application Programming Interface
(HE-API) [329] has currently integrated HELib and FV-NFLib [99], an implement-
mentation of the Fan and Vercauteren (FV) [129] SHE scheme, under a single
API. The authors appear to have made significant improvements in circuit eval-
uation times, but few details have been made available [57]. Microsoft’s SEAL
library \[198\] also implements the FV scheme, albeit, in a modified form. FHEW \[117\] implements the FHE scheme described in \[116\]. The performance of these implementations is discussed in section \[7.7\].

Erkin et al. \[128\] exploit the linearly-homomorphic properties of Paillier to compute feature vector matches in their privacy-preserving face recognition algorithm. Our schemes can likewise compute known linear functions, simply by not encrypting the coefficients of the function.

Catalano et al. \[74\] aim to extend a linearly-homomorphic system, such as Paillier \[262\], to compute multivariate quadratics homomorphically. However, their extension relies on pre-computing a product for each pair of plaintexts and then applying a linear function on the encryption of these products. As such, it does not extend the underlying linear encryption scheme and is not multiplicatively homomorphic. They claim that their system can compute any degree 2 polynomial with at most one multiplication. However, it is not clear how they would compute the polynomial \[m_1 \cdot (m_2 + \ldots + m_n)\] without performing \(n-1\) offline multiplications. By contrast, our scheme would only require one multiplication. In \[73\], Catalano et al. extend their approach to cubics.

Zhou and Wornell \[365\] construct a scheme based on integer vectors, similar, in some respects, to our HE2 (section 7.3.1) and HE \[7.4\] schemes. Bogos et al. \[51\] demonstrate that the system displays some theoretical insecurities. However, the question of whether these are of practical importance is not addressed.

The symmetric MORE scheme \[191\] uses linear transformations, as do our schemes but in a different way. MORE has been shown \[330\] to be insecure against KPA, at least as originally proposed. However, whether KPA is relevant in applications of the scheme is unclear.

Recent work on functional encryption \[149\] should also be noted. While these results are of great theoretical interest, the scenario where such schemes might be applied is rather different from our model. Also, the methods of \[149\] seem too computationally expensive to be of practical interest in the immediate future.

We also note the work of Cheon et al. \[82\]. They use the Chinese Remainder Theorem (CRT) in an HE system. We make use of the CRT in our scheme HE2NCRT below (section 7.5). However, our construction differs significantly
from theirs.

# 7.2 Initial Homomorphic Scheme

In this section we present details of our initial SHE schemes over the integers.

## 7.2.1 Sufficient Entropy (HE1)

We have integer inputs $m_1, m_2, \ldots, m_n \in [0, M)$. (Negative integers can be handled as in van Dijk et al. \[111\], by taking residues in $[-(p - 1)/2, (p - 1)/2)$, rather than $[0, p)$.) We wish to compute a polynomial $P$ of degree $d$ in these inputs. The inputs are distributed with entropy $\rho$, where $\rho$ is large enough, as discussed in section 7.1.1 above. In practical terms, $\rho \geq 32$ will provide sufficient entropy for strong security, since breaking the system would require more than a billion guesses. Our HE scheme is the system ($KGen, Pgen, Enc, Dec, Add, Mult$).

### Key and Parameter Generation.

Let $\lambda$ be a security parameter, measured in bits. Let $p$ and $q$ be randomly chosen large distinct primes such that $p \in [2^{\lambda - 1}, 2^{\lambda}]$, and $q \in [2^{\eta - 1}, 2^\eta]$, where $\eta \approx \lambda^2/\rho - \lambda$. Here $\lambda$ must be large enough to negate direct factorisation of $pq$ (see \[192\]), and $p$ and $q$ are chosen to negate Coppersmith’s attack \[93\]. We will also require $p > (n + 1)^d M^d$ to ensure that $P(m_1, m_2, \ldots, m_n) < p$, so that the result of the computation can be successfully decrypted. Our bounds are worst case, allowing for polynomials which contain all possible monomial terms. For some applications, they will be much larger than required to ensure that $P(m_1, m_2, \ldots, m_n) < p$ and smaller bounds will suffice. Our algorithms $KGen$ (Algorithm 7.1) and $Pgen$ (Algorithm 7.2) will randomly select $p$ and $q$ according to these bounds. Then $p$ is the private symmetric key for the system and $pq$ is the modulus for arithmetic performed by Add and Mult. $pq$ is a public parameter of the system. We assume that the entropy $\rho \gg \lg \lambda$, so that a brute force attack cannot be carried out in polynomial time.
Algorithm 7.1: KGen: Key Generation Algorithm

Input : $\lambda \in S$
Output: $p \in K$: secret key
1. $p \leftarrow [2^{\lambda-1}, 2^\lambda]$
2. return $p$

Algorithm 7.2: Pgen: Parameter Generation Algorithm

Input : $\lambda \in S$
Input : $\rho \in \mathbb{Z}$: entropy of inputs
Input : $p$: secret key
Output: modulus $\in \mathbb{Z}$: public modulus
1. $\eta \leftarrow \lambda^2/\rho - \lambda$
2. $q \leftarrow [2^{\eta-1}, 2^\eta]$
3. modulus $\leftarrow pq$
4. return modulus

Security parameters.

We can easily set the security parameters $\lambda$ and $\eta$ to practical values. If $n \approx \sqrt{M}$, $M \approx 2^\rho$ then we may take $\lambda \approx 3d\rho/2$ and $\eta \approx 3d\lambda/2 - \lambda$ (see appendix C). For example, if $\rho = 32$, $d = 4$, we can take any $\lambda > 192$, $\eta > 960$.

Encryption.

We encrypt a plaintext integer $m$ using Enc(Algorithm 7.3).

Algorithm 7.3: Enc: Encryption algorithm

Input : $m \in M$
Input : $p$: secret key
Input : modulus: public modulus
Output: $c \in C$
1. $q \leftarrow \text{modulus}/p$
2. $r \leftarrow [1, q]$
3. $c \leftarrow m + rp \pmod{\text{modulus}}$
4. return $c$
Decryption.

We decrypt the ciphertext $c$ using $\text{Dec}$ (Algorithm 7.4).

**Algorithm 7.4: Dec: Decryption algorithm**

<table>
<thead>
<tr>
<th>Input</th>
<th>$c \in C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>$p$: secret key</td>
</tr>
<tr>
<td>Output</td>
<td>$m \in M$</td>
</tr>
<tr>
<td>1</td>
<td>$m \leftarrow c \pmod{p}$</td>
</tr>
<tr>
<td>2</td>
<td>return $m$</td>
</tr>
</tbody>
</table>

Addition.

The sum modulo $pq$ of two ciphertexts, $c = m + rp$ and $c' = m' + r'p$, is given by $\text{Add}$ (Algorithm 7.5). Since $\text{Add}(c, c') = c + c' = m + m' + (r + r')p$, $\text{Add}(c, c')$ decrypts to $m + m'$, provided $m + m' < p$.

**Algorithm 7.5: Add: addition algorithm**

<table>
<thead>
<tr>
<th>Input</th>
<th>$c \in C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>$c' \in C$</td>
</tr>
<tr>
<td>Input</td>
<td>modulus $\in \mathbb{Z}$: public modulus</td>
</tr>
<tr>
<td>Output</td>
<td>result $\in C$</td>
</tr>
<tr>
<td>1</td>
<td>result $\leftarrow c + c' \pmod{\text{modulus}}$</td>
</tr>
<tr>
<td>2</td>
<td>return result</td>
</tr>
</tbody>
</table>

Multiplication

The product modulo $pq$ of two ciphertexts, $c = m + rp$ and $c' = m' + r'p$, is given by $\text{Mult}$ (Algorithm 7.6). Since $\text{Mult}(c, c') = cc' = mm' + (rm' + r'm + rr'p)p$, it decrypts to $mm'$, provided $mm' < p$.

Security.

Security of the system is provided by the *partial approximate common divisor problem* (PACDP), first posed by Howgrave-Graham [164], but can be formulated [78, 92] as:
**Algorithm 7.6: Mult: multiplication algorithm**

<table>
<thead>
<tr>
<th>Input</th>
<th>$c \in \mathbb{C}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>$c' \in \mathbb{C}$</td>
</tr>
<tr>
<td>Input</td>
<td>modulus $\in \mathbb{Z}$: public modulus</td>
</tr>
<tr>
<td>Output</td>
<td>result $\in \mathbb{C}$</td>
</tr>
</tbody>
</table>

1. result $\leftarrow cc' \pmod{\text{modulus}}$
2. return result

**Definition 20. (Partial approximate common divisor problem.)** Suppose we are given one input $x_0$, of the form $pr_0$, and $n$ inputs $x_i$, of the form $pr_i + m_i$, $i \in [1, n]$, where $p$ is an unknown constant integer and the $m_i$ and $r_i$ are unknown integers. We have a bound $B$ such that $|m_i| < B$ for all $i$. Under what conditions on the $m_i$ and $r_i$, and the bound $B$, can an algorithm be found that can uniquely determine $p$ in time polynomial in the total bit length of the numbers involved?

A straightforward attack on this problem is by brute force. Consider $x_1$. Assuming that $m_1$ is sampled from $D$, having entropy $\rho$, we successively try values for $m_1$ and compute $\gcd(x_0, x_1 - m_1)$ in polynomial time until we find a divisor that is large enough to recover $p$. Then we can recover $m_i$ as $(x_i \mod p)$ for $i \in [2, n]$. As discussed in section 7.1.1, the search will require $2^\rho \gcd$ operations in expectation. Note that publicly known constants, need not, and should not be encrypted. Encrypting them provides an obvious guessing attack.

Several attempts have been made to solve the PACDP [78, 92, 164], resulting in theoretically faster algorithms for some cases of the problem. The paper [78] gives an algorithm requiring only $\sqrt{M}$ polynomial time operations if $D$ is the uniform distribution on $[0, M)$, and hence $\rho = \log M$. No algorithm running in time subexponential in $\rho$ is known for this problem, so the encryption will be secure if $\rho$ is large enough. See [133] for a survey and evaluation of attacks on PACDP. We also note the work of Cheon and Stehlé [80] which shows that Regev’s “learning with errors” (LWE) problem [281] can be reduced to the approximate common divisor problem (ACDP), demonstrating that ACDP is at least as hard as LWE. As discussed in 3.2.1 is the basis of many lattice based FHE schemes.

Our system is a special case of PACDP, since we use the residues modulo a distinct semiprime. A semiprime is a natural number that is the product of two primes. A distinct semiprime is a semiprime where the primes are distinct. We call this
the *semiprime partial approximate common divisor problem* (SPACDP). It is a restriction, but there is no reason to believe that it is any easier than PACDP.

**Definition 21** (Semiprime factorisation problem.). Given a semiprime $s$, the product of primes $p$ and $q$, can $p$ and $q$ be determined in polynomial time?

The computational complexity of this problem, which lies at the heart of the widely-used RSA cryptosystem, is open, other than for quantum computing, which currently remains impractical. We will show that breaking HE1 is equivalent to semiprime factorisation. Therefore, our scheme is at least as secure as unpadded RSA \[285\].

**Theorem 1.** An attack against HE1 is successful in polynomial time if and only if we can factorise a distinct semi-prime in polynomial time.

The proof of this theorem, and proofs of all other theorems and lemmas, can be found in Appendix B.

With low entropy plaintexts, there is a brute force attack on this system, which we call a collision attack. Suppose we have a pair of equal plaintexts $m_1 = m_2$. The difference between their encryptions $(c_1 - c_2)$ is an encryption of 0, and KPA is possible. In fact, for $n$ plaintexts $m_1, m_2, \ldots, m_n$, if there exist $i, j \in [1, n]$ with $m_i = m_j$, then $\prod_{1 \leq i < j \leq n} (c_j - c_i)$ is an encryption of 0. However, if there is sufficient entropy, this attack is not possible.

**Lemma 2.** If the inputs $m$ have entropy $\rho$ then, for any two independent inputs $m_1, m_2$, $\Pr(m_1 = m_2) \leq 2^{-\rho}$.

Thus, for $n$ inputs, $m_1, m_2, \ldots, m_n$ the probability that there exist $i, j \in [1, n]$ with $m_i = m_j$ is at most $\binom{n}{2} 2^{-\rho}$. If $n < 2^{-\rho/3}$, this probability is at most $2^{-\rho/3}$. Hence, for large enough $\lambda$, collision attack is infeasible.

Although this outside the scope of our usage scenario, we should note that, if HE1 is used for an interactive computation or for computation on a persistent data source, it is vulnerable to CPA. If, for example, our scheme was used to encrypt a column in a persistent database, an attacker could create two records containing the same plaintext integer value. He can then use the “collision” attack described above to obtain an encryption of 0 and then obtain the secret key by finding the GCD of this value and the public modulus. If such attacks are envisaged, the “N” versions should be used, which are more resistant.
7.2.2 Insufficient Entropy (HE1N)

Suppose now that the integer inputs \( m_i, i \in [1, n] \), are distributed with entropy \( \rho \), where \( \rho \) is not large enough to negate a brute force guessing attack. Therefore, we increase the entropy of the plaintext by adding an additional “noise” term to the ciphertext. This will be a multiple \( s \) (from 0 to \( \kappa \)) of an integer \( \kappa \), chosen so that the entropy \( \rho' = \rho + \lg \kappa \) is large enough to negate a brute force guessing attack. As a result of the extra linear term in the ciphertext, we compute \( P(m_1, \ldots, m_n, \kappa) \) instead. We can easily retrieve \( P(m_1, \ldots, m_n) \) from \( P(m_1, \ldots, m_n, \kappa) \).

Key and Parameter Generation.

\( \text{KGen} \) (Algorithm 7.7) and \( \text{Pgen} \) (Algorithm 7.7) now randomly choose \( p \) and \( q \) as in HE1, but with \( \eta = \lambda^2/\rho' - \lambda \), and \( p > (n + 1)^d(M + \kappa^2)^d \) so that \( P(m_1 + s_1\kappa, m_2 + s_2\kappa, \ldots, m_N + s_n\kappa) < p \), when \( s_1, s_2, \ldots, s_n \in [0, \kappa) \). \( \text{KGen} \) also randomly chooses \( \kappa \), where \( \kappa > (n + 1)^dM^d \), so that \( P(m_1, m_2, \ldots, m_n) < \kappa \). The secret key, \( \text{sk} \), is now \( (\kappa, p) \).

**Algorithm 7.7: KGen: Key Generation Algorithm**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda \in \mathcal{S} )</td>
<td>( (\kappa, p) ): secret key</td>
</tr>
<tr>
<td>( \rho \in \mathbb{Z} ): entropy of input</td>
<td></td>
</tr>
<tr>
<td>( \rho' \in \mathbb{Z} ): effective entropy of inputs</td>
<td></td>
</tr>
<tr>
<td>( 1 \ p \leftarrow [2^{\lambda - 1}, 2^\lambda] )</td>
<td>( 2 \ \nu \leftarrow \rho' - \lambda )</td>
</tr>
<tr>
<td>( 3 \ \kappa \leftarrow [2^{\nu - 1}, 2^\nu] )</td>
<td>( 4 \ \text{return} \ (\kappa, p) )</td>
</tr>
</tbody>
</table>

**Algorithm 7.8: Pgen: Parameter Generation Algorithm**

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda \in \mathcal{S} )</td>
<td>( \eta \in \mathbb{Z} ): modulus for arithmetic</td>
</tr>
<tr>
<td>( \rho' \in \mathbb{Z} ): effective entropy of inputs</td>
<td>( \eta = \lambda^2/\rho' - \lambda )</td>
</tr>
<tr>
<td>( (\kappa, p) ): secret key</td>
<td>( 2 \ q \leftarrow [2^{\eta - 1}, 2^\eta] )</td>
</tr>
<tr>
<td>( \text{modulus} \in \mathbb{Z} ): modulus for arithmetic</td>
<td>( 3 \ \text{modulus} \leftarrow pq )</td>
</tr>
<tr>
<td>( 4 \ \text{return} \ \text{modulus} )</td>
<td></td>
</tr>
</tbody>
</table>
7.2. INITIAL HOMOMORPHIC SCHEME

Security parameters.

Again, we can set the security parameters $\lambda$ and $\eta$ to practical values. If we assume $M \approx 2^\rho$ and large enough $n$, as in section 7.2.1 then we may take $\lg \kappa > d(\lg n + \rho), \rho' = \rho + \lg \kappa, \lambda > d(\lg n + 2 \lg \kappa)$. Then, for example, if $d = 3$, $\lg n = 16, \rho = 8$, then $\lg \kappa > 72, \rho' = 80, \lambda > 480, \eta > 2400$. In the extreme case that the inputs are bits, so $\rho = 1$, and $d = 3, \lg n = 16$, then we can take $\lg \kappa \approx 51$ and $\rho' \approx 52$, and we have $\lambda > 354, \eta > 2056$, which is only 15% smaller than for $\rho = 8$.

Encryption.

We encrypt plaintext $m$ using $\text{Enc}(\text{Algorithm 7.9})$.

<table>
<thead>
<tr>
<th>Algorithm 7.9: Enc: Encryption Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: $m \in \mathcal{M}$</td>
</tr>
<tr>
<td><strong>Input</strong>: $(\kappa, p)$: secret key</td>
</tr>
<tr>
<td><strong>Input</strong>: modulus: public modulus</td>
</tr>
<tr>
<td><strong>Output</strong>: $c \in \mathcal{C}$</td>
</tr>
<tr>
<td>1 $q \leftarrow \text{modulus}/p$</td>
</tr>
<tr>
<td>2 $r \leftarrow [1, q]$</td>
</tr>
<tr>
<td>3 $s \leftarrow [0, \kappa]$</td>
</tr>
<tr>
<td>4 $c \leftarrow m + sk + rp \mod \text{modulus}$</td>
</tr>
<tr>
<td>5 return $c$</td>
</tr>
</tbody>
</table>

Decryption.

We decrypt ciphertext $c$ using $\text{Dec}(\text{Algorithm 7.10})$.

<table>
<thead>
<tr>
<th>Algorithm 7.10: Dec: Decryption Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: $c \in \mathcal{C}$</td>
</tr>
<tr>
<td><strong>Input</strong>: $(\kappa, p)$: secret key</td>
</tr>
<tr>
<td><strong>Output</strong>: $m \in \mathcal{M}$</td>
</tr>
<tr>
<td>1 $m \leftarrow (c \mod p) \mod \kappa$</td>
</tr>
<tr>
<td>2 return $m$</td>
</tr>
</tbody>
</table>
Arithmetic.

Addition and multiplication of ciphertexts is given by Algorithms 7.5 and 7.6.

Security.

The use of random noise gives the encryption the following “indistinguishability” property, which implies that the system satisfies IND-CPA [28, 35].

**Theorem 3.** For any encryption $c$, $c \mod \kappa$ is polynomial time indistinguishable from the uniform distribution on $[0, \kappa)$. Thus HE1N satisfies IND-CPA, under the assumption that SPACDP is not polynomial time solvable.

Therefore, HE1N is resistant to both the “guessing” and “collision” attacks discussed in section 7.2.1.

Hybrid scheme.

Note that mixed data, some of which has high entropy and some low, can be encrypted with a hybrid of HE1 and HE1N. More generally, we can choose $s$ to be smaller for higher entropy and larger for lower entropy, say $s \in [0, \chi_i)$, where $0 \leq \chi_i < \kappa$, for the $i$th data type, rather than $[0, \kappa)$. However, $\kappa$ itself remains the same for all $i$, or we cannot decrypt. Then the entropy increases to $\rho_i + \lg \chi_i$ for data type $i$. The advantage is a smaller blow-up in the noise. A possible disadvantage is that this mixed scheme may not necessarily have the IND-CPA property of Theorem 3. The same idea can be applied to HE2 and HE2N below, and to the HE$k$N schemes, for $k > 2$, described in section 7.4.2.

Again, like HE1, if HE2 is used for an interactive computation or for computation on a persistent data source, it is vulnerable to CPA using the “collision” attack described in section 7.2.1.

7.3 Adding a dimension

In this section we discuss adding an additional dimension to the ciphertext, which becomes a 2-vector. The purpose of this is to increase the level of security beyond
HE1 and HE1N. In both schemes presented below, HE2 and HE2N, we add a further vector term, with two further secret parameters. The two schemes presented below have a constant factor overhead for arithmetic operations. An addition operation in the plaintext space requires two additions in the ciphertext space, and a multiplication in the plaintext space requires nine multiplications and four additions in the ciphertext space.

### 7.3.1 Sufficient entropy (HE2)

As with HE1, it is assumed that the inputs $m_i (i \in [1, n])$ are of sufficient entropy.

#### Key and Parameter Generation.

$p$ and $q$ are randomly chosen by $KGen$ (Algorithm 7.11) according to the bounds given in section 7.2.1. $KGen$ sets $\mathbf{a} = [a_1 \ a_2]^T$, where $a_i \xleftarrow{\$} [1, pq]$ ($i \in [1, 2]$) such that $a_1, a_2, a_1 - a_2 \neq 0 \pmod{p \text{ and mod } q}$.

<table>
<thead>
<tr>
<th>Algorithm 7.11: KGen: Key Generation Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input : $\lambda \in S$</td>
</tr>
<tr>
<td>Input : $\rho \in \mathbb{Z}$: entropy of inputs</td>
</tr>
<tr>
<td>Output: $(p, \mathbf{a})$: secret key</td>
</tr>
<tr>
<td>Output: modulus: public modulus</td>
</tr>
<tr>
<td>1 $\ p \xleftarrow{$} [2^{\lambda-1}, 2^\lambda]$</td>
</tr>
<tr>
<td>2 $\eta \leftarrow \lambda^2 / \rho - \lambda$</td>
</tr>
<tr>
<td>3 $\ q \xleftarrow{$} [2^{\eta-1}, 2^{\eta}]$</td>
</tr>
<tr>
<td>4 modulus $\leftarrow pq$</td>
</tr>
<tr>
<td>5 repeat</td>
</tr>
<tr>
<td>6 $\quad a_i \xleftarrow{$} [1, \text{modulus}]$ ($i \in [1, 2]$)</td>
</tr>
<tr>
<td>7 until $a_1, a_2, a_1 - a_2 \neq 0$ (mod $p \text{ and mod } q$)</td>
</tr>
<tr>
<td>8 $\mathbf{a} \leftarrow [a_1 \ a_2]^T$</td>
</tr>
<tr>
<td>9 return $(p, \mathbf{a}), \text{modulus}$</td>
</tr>
</tbody>
</table>

$Pgen$(Algorithm 7.12) generates $R$, the re-encryption matrix (see “Multiplication”).

---

2 The condition $a_1, a_2, a_1 - a_2 \neq 0 \pmod{p \text{ and mod } q}$ fails with exponentially small probability $3(1/p+1/q)$. Thus, $a_1$ and $a_2$ are indistinguishable in polynomial time from $a_1, a_2 \xleftarrow{\$} [0, pq]$. 
### Algorithm 7.12: Pgen: Parameter Generation Algorithm

<table>
<thead>
<tr>
<th>Input</th>
<th>(p, a): secret key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>modulus: public modulus</td>
</tr>
<tr>
<td>Output</td>
<td>R: public re-encryption matrix</td>
</tr>
</tbody>
</table>

1. \( q \leftarrow \text{modulus}/p \)
2. \( \beta \leftarrow 2(a_2 - a_1)^2 \)
3. \( g \leftarrow [0, q] \)
4. \( \sigma \leftarrow [0, \text{modulus}] \)
5. \( \alpha_1 \leftarrow \beta^{-1}(\sigma a_1 + gp - a_1^2) \)
6. \( \alpha_2 \leftarrow \beta^{-1}(\sigma a_2 + gp - a_2^2) \)
7. \( R \leftarrow \begin{bmatrix} 1 - 2\alpha_1 & \alpha_1 & \alpha_1 \\ -2\alpha_2 & \alpha_2 + 1 & \alpha_2 \end{bmatrix} \)
8. return \( R \)

### Encryption.

We encrypt a plaintext integer \( m \) as the 2-vector \( c \) using \( \text{Enc}(\text{Algorithm 7.13}) \), where \( \mathbf{1} = [1 \ 1]^T \) \( r \) and \( s \) are independent. We note that two encryptions of the same plaintext are different with very high probability.

### Theorem 4.
The encryption scheme produces ciphertexts with components which are random integers modulo \( pq \).

Note, however, that the components of the ciphertexts are correlated, and this may be a vulnerability. We discuss this later in this section (“Cryptanalysis”).
Decryption.

To decrypt, we use Algorithm 7.14. We call \( \gamma \) the \textit{decryption vector}.

\textbf{Algorithm 7.14: Dec: Decryption Algorithm}

\begin{align*}
\text{Input} & : c \in \mathbb{C} \\
\text{Input} & : (p, a) \text{: secret key} \\
\text{Output} & : m \in \mathbb{M} \\
1 & \gamma^T \leftarrow (a_2 - a_1)^{-1}[a_2 - a_1] \\
2 & m \leftarrow \gamma^T c \mod p \\
3 & \text{return } m
\end{align*}

Addition.

We define the addition operation on ciphertexts as the vector sum modulo \( pq \) of the two ciphertext vectors \( c \) and \( c' \).

\textbf{Algorithm 7.15: Add: addition algorithm}

\begin{align*}
\text{Input} & : c \in \mathbb{C} \\
\text{Input} & : c' \in \mathbb{C} \\
\text{Input} & : \text{modulus} \in \mathbb{Z} \text{: modulus for arithmetic} \\
\text{Output} & : \text{result} \in \mathbb{C} \\
1 & \text{result} \leftarrow c + c' \mod \text{modulus} \\
2 & \text{return } \text{result}
\end{align*}

Therefore, if inputs \( m, m' \) encrypt as \((m + rp)1 + sa, (m' + r'p+)1 + s'a\),

\[
\text{Add}(c, c') = c + c' = (m + m' + (r + r')p)1 + (s + s')a.
\]

which is a valid encryption of \( m + m' \).

Multiplication.

If \( c = [c_1 \ c_2]^T \), we construct the augmented ciphertext vector, \( c_* = [c_1 \ c_2 \ c_3]^T \), where \( c_3 = 2c_1 - c_2 \). Thus, \( c_3 = (m + rp) + sa_3 \mod pq \), for \( a_3 = 2a_1 - a_2 \). So,

\[
\text{Mult}(c, c') = c \cdot c' = R(c_* \circ c_*') \mod pq,
\]
Algorithm 7.16: Mult: multiplication algorithm

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( c_3 \leftarrow 2c_1 - c_2 )</td>
</tr>
<tr>
<td>2</td>
<td>( c_* \leftarrow [c_1 \ c_2 \ c_3]^T )</td>
</tr>
<tr>
<td>3</td>
<td>( c'_3 \leftarrow 2c'_1 - c'_2 )</td>
</tr>
<tr>
<td>4</td>
<td>( c'_* \leftarrow [c'_1 \ c'_2 \ c'_3]^T )</td>
</tr>
<tr>
<td>5</td>
<td>result ( \leftarrow R(c_* \circ c'_*) ) (mod modulus)</td>
</tr>
<tr>
<td>6</td>
<td>return result</td>
</tr>
</tbody>
</table>

where \( \cdot \) is a product on \( \mathbb{Z}_{pq}^2 \) and \( c_* \circ c'_* \) is the Hadamard product modulo \( pq \) of the two augmented ciphertext vectors \( c_* \) and \( c'_* \).

**Theorem 5.** If \( c \) is an encryption of \( m \) and \( c' \) is an encryption of \( m' \) then \( R(c_* \circ c'_*) \) (mod \( pq \)) is an encryption of \( mm' \).

Observe that \( \alpha_1, \alpha_2 \) in \( R \) are public, but give only two equations for the four parameters of the system \( a_1, a_2, \sigma, \rho p \). These equations are quadratic mod \( pq \), and solving them is as hard as semiprime factorisation in the worst case \[276\].

Also, observe that, independently of \( a \),

\[
Rc_* = (m + rp)R1_* + sRa_* = (m + rp)1 + sa = c,
\]

for any ciphertext \( c \). Hence re-encrypting a ciphertext gives the identity operation, and discloses no information.

**Hardness.**

We can show that this system is at least as hard as SPACDP. In fact,

**Theorem 6.** SPACDP is of equivalent complexity to the special case of HE2 where \( \delta = a_2 - a_1 \ (0 < \delta < p) \) is known.

Without knowing the parameter \( \delta = a_2 - a_1 \), HE2 cannot be reduced to SPACDP in this way, so HE2 is more secure than HE1.
Cryptanalysis.

Each new ciphertext $c$ introduces two new unknowns $r, s$ and two equations for $c_1, c_2$. Thus we gain no additional information from a new ciphertext. However, if we can guess, $m, m'$ for any two ciphertexts $c, c'$, we can determine

$$(c_1 - m) = rp + sa_1, \quad (c_2 - m) = rp + sa_2,$$
$$(c'_1 - m') = r'p + s'a_1, \quad (c'_2 - m') = r'p + s'a_2,$$
$$(c_1 - m)(c'_2 - m') - (c_2 - m)(c'_1 - m') = (a_2 - a_1)(rs' - r's)p \pmod{pq}$$

Since $a_2 \neq a_1$, and $sr' \neq s'r$ with high probability, this is a nonzero multiple of $p, \nu p$ say. We may assume $\nu < q$, so $p = \gcd(\nu p, pq)$. We can now solve the linear system $\gamma^T[c \ c'] = [m \ m'] \pmod{p}$ to recover the decryption vector. This effectively breaks the system, since we can now decrypt an arbitrary ciphertext. We could proceed further, and attempt to infer $a_1$ and $a_2$, but we will not do so.

Note that to break this system, we need to guess two plaintexts, as opposed to one in HE1. The entropy of a pair $(m, m')$ is $2\rho$, so we have effectively squared the number of guesses needed to break the system relative to HE1. So HE2 can tolerate smaller entropy than HE1. We note further that HE2 does not seem immediately vulnerable to known cryptanalytic attacks on HE1 \cite{78, 92, 164}.

### 7.3.2 Insufficient entropy (HE2N)

In this section we extend HE1N above (section 7.2.2) to two dimensions.

**Key and Parameter Generation.**

$\text{KGen}$ (Algorithm 7.17) randomly chooses $p, q$ and $\kappa$ according to the bounds given in section 7.2.2. $R$ is generated by $\text{Pgen}$ (which exactly the same as Algorithm 7.12). The secret key is $(\kappa, p, a)$, and the public parameters are $pq$ and $R$, defined in section 7.3.1.
**Algorithm 7.17: KGen: Key Generation Algorithm**

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\nu \leftarrow \rho' - \rho$</td>
</tr>
<tr>
<td>2</td>
<td>$\kappa \leftarrow [2^{\nu-1}, 2^\nu]$</td>
</tr>
<tr>
<td>3</td>
<td>$p \leftarrow [2^{\lambda-1}, 2^{\lambda}]$</td>
</tr>
<tr>
<td>4</td>
<td>$\eta \leftarrow \lambda^2/\rho' - \lambda$</td>
</tr>
<tr>
<td>5</td>
<td>$q \leftarrow [2^{\eta-1}, 2^\eta]$</td>
</tr>
<tr>
<td>6</td>
<td>modulus $\leftarrow pq$</td>
</tr>
<tr>
<td>7</td>
<td>repeat</td>
</tr>
<tr>
<td>8</td>
<td>$a_i \leftarrow [1, \text{modulus}]$ (if $i \in [1, 2]$)</td>
</tr>
<tr>
<td>9</td>
<td>until $a_1, a_2, a_1 - a_2 \neq 0$ (mod $p$ and mod $q$)</td>
</tr>
<tr>
<td>10</td>
<td>$a \leftarrow [a_1, a_2]^T$</td>
</tr>
<tr>
<td>11</td>
<td>return $(\kappa, p, a), \text{modulus}$</td>
</tr>
</tbody>
</table>

**Algorithm 7.18: Enc: Encryption Algorithm**

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$q \leftarrow \text{modulus}/p$</td>
</tr>
<tr>
<td>2</td>
<td>$r \leftarrow [0, q]$</td>
</tr>
<tr>
<td>3</td>
<td>$s \leftarrow [0, \kappa]$</td>
</tr>
<tr>
<td>4</td>
<td>$t \leftarrow [0, \text{modulus}]$</td>
</tr>
<tr>
<td>5</td>
<td>$c \leftarrow (m + rp + s\kappa)1 + t\alpha \pmod{\text{modulus}}$</td>
</tr>
<tr>
<td>6</td>
<td>return $c$</td>
</tr>
</tbody>
</table>
7.4. GENERALISATION TO K DIMENSIONS

Encryption.

We encrypt a plaintext integer \( m \in [0, M) \) as a 2-vector \( c \), using Enc (Algorithm 7.18). \( 1 \) is defined as in section 7.3.1.

Decryption.

\[
\begin{align*}
\text{Algorithm 7.19: Dec: Decryption Algorithm} \\
\text{Input} & : c \in C \\
\text{Input} & : (\kappa, p, a) \text{: secret key} \\
\text{Output} & : m \in M \\
1 & \quad \gamma^T \leftarrow (a_2 - a_1)^{-1}[a_2 - a_1] \\
2 & \quad m \leftarrow (\gamma^T c \mod p) \mod \kappa \\
3 & \quad \text{return } m
\end{align*}
\]

We decrypt a ciphertext \( c \) using Dec (Algorithm 7.19).

Arithmetic.

Addition and multiplication of ciphertexts are performed by algorithms Add and Mult (which are exactly the same as Algorithms 7.5 and 7.16).

Security.

HE2N has all the properties of HE1N. However, it is more secure, since there is an additional unknown parameter in the ciphertext. We also note that HE2N satisfies Theorem 3, so it inherits the IND-CPA property.

7.4 Generalisation to \( k \) dimensions

In this section, we generalise HE2 and HE2N to \( k \)-vectors. HE1 and HE1N are the cases for \( k = 1 \) and HE2 and HE2N are the cases for \( k = 2 \).
7.4.1 Sufficient entropy (HE$k$)

We generalise HE2 to $k$ dimensions. We extend our definition of an augmented vector $v_*$, for a $k$-vector, $v$, such that $v_*$ is a $(k+1)$-vector, with components $v_i$ ($1 \leq i \leq k$) followed by $2v_i - v_j$ ($1 \leq i < j \leq k$). For example, if $k = 3$, then $v_* = [v_1, v_2, v_3, 2v_1 - v_2, 2v_1 - v_3, 2v_2 - v_3]$. In general, for $\ell > k$, $v_\ell = 2v_i - v_j$, where $\ell = \binom{i}{2} + k + j - 1$. Note that $v_* = U_k v$ for a $(k+1) \times k$ matrix with entries $0, \pm 1, 2$, and whose first $k$ rows form the $k \times k$ identity matrix $I_k$. For example, if $k = 3$,

$$U_3 = \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
2 & -1 & 0 \\
2 & 0 & -1 \\
0 & 2 & -1
\end{bmatrix}$$

Note that $v_* = U_k v$ implies that $1_*$ is the $(\binom{k}{2})$ vector of 1’s, and that $*$ is a linear mapping, i.e. $(r_1 v_1 + r_2 v_2)_* = r_1 v_1* + r_2 v_2*$.

Key Generation

**Algorithm 7.20: KGen: Key Generation Algorithm**

<table>
<thead>
<tr>
<th>Input</th>
<th>$\lambda \in S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>$\rho \in \mathbb{Z}$: entropy of inputs</td>
</tr>
<tr>
<td>Output</td>
<td>$(p, a_1, \ldots, a_{k-1})$: secret key</td>
</tr>
<tr>
<td>Output</td>
<td>modulus: public modulus</td>
</tr>
</tbody>
</table>

1. $p \leftarrow \mathbb{P}[2\lambda - 1, 2\lambda]$
2. $\eta \leftarrow \lambda^2 / \rho - \lambda$
3. $q \leftarrow \mathbb{P}[2^\eta - 1, 2^\eta]$
4. modulus $\leftarrow pq$
5. $a_0 \leftarrow 1$
6. repeat
7. $a_j \leftarrow [1, \text{modulus}]^k, j \in [1, k)$
8. until $a_j, j \in [0, k)$, form a basis for $\mathbb{Z}_{pq}^k$
9. return $(p, a_1, \ldots, a_{k-1}), \text{modulus}$

KGen is detailed in Algorithm 7.20. It randomly chooses $p$ and $q$ according to
the bounds given in section 7.3.1. In addition to choosing \( p \) and \( q \), \( K\text{Gen} \) also randomly chooses vectors \( a_1, \ldots, a_{k-1} \) such that \( a_0, a_1, \ldots, a_{k-1} \) form a basis for \( \mathbb{Z}^k_{pq} \), where \( a_0 = 1 \), the \( k \)-vector whose elements are all 1. We denote \( A_k \) as the \( k \times k \) matrix \([a_0 \ a_1 \ldots a_{k-1}]\) where the columns of \( A_k \) are the \( a_i, i \in [0, k) \). We show that the probability that \( a_1, \ldots, a_{k-1} \) do not form a basis is negligible for large \( p \) and \( q \) in Lemma 7.

**Lemma 7.** \( \Pr(a_0, a_1, \ldots, a_{k-1} \text{ do not form a basis for } \mathbb{Z}^k_{pq}) \leq (k-1)(1/p+1/q) \).

### Parameter Generation

**Algorithm 7.21: Pgen: Parameter Generation Algorithm**

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( q \leftarrow \text{modulus}/p )</td>
</tr>
<tr>
<td>2</td>
<td>( a_0 \leftarrow 1 )</td>
</tr>
<tr>
<td>3</td>
<td>( A_k \leftarrow [a_0 \ a_1 \ldots a_{k-1}] )</td>
</tr>
<tr>
<td>4</td>
<td>foreach ( j \in [0, k) ) do</td>
</tr>
<tr>
<td>5</td>
<td>( a_{*j} \leftarrow U_k a_j )</td>
</tr>
<tr>
<td>6</td>
<td>end</td>
</tr>
<tr>
<td>7</td>
<td>foreach ( i, j \in [0, k), 0 \leq i \leq j &lt; k ) do</td>
</tr>
<tr>
<td>8</td>
<td>( a^2_{*,ij} \leftarrow a_{*i} \circ a_{*j} )</td>
</tr>
<tr>
<td>9</td>
<td>end</td>
</tr>
<tr>
<td>10</td>
<td>( A^2_{<em>k} \leftarrow [a^2_{</em>,0} \ldots a^2_{<em>,ij} \ldots a^2_{</em>,k-2,k-1}] ) // ( A^2_{*k} ) is a ((k+1)^2\times(k+1)^2) matrix</td>
</tr>
<tr>
<td>11</td>
<td>if ( A^2_{*k} ) has no inverse then</td>
</tr>
<tr>
<td>12</td>
<td>( \text{return } \perp )</td>
</tr>
<tr>
<td>13</td>
<td>end</td>
</tr>
<tr>
<td>14</td>
<td>foreach ( i, j \in [0, k), 0 \leq i \leq j &lt; k ) do</td>
</tr>
<tr>
<td>15</td>
<td>( \theta_{ij} \leftarrow [0, q) )</td>
</tr>
<tr>
<td>16</td>
<td>foreach ( l \in [1, k) ) do</td>
</tr>
<tr>
<td>17</td>
<td>( \sigma_{ijl} \leftarrow [0, \text{modulus}) )</td>
</tr>
<tr>
<td>18</td>
<td>end</td>
</tr>
<tr>
<td>19</td>
<td>( b_{ij} \leftarrow \theta_{ij} p 1 + \sum_{l=1}^{k-1} \sigma_{ijl} a_l )</td>
</tr>
<tr>
<td>20</td>
<td>end</td>
</tr>
<tr>
<td>21</td>
<td>( C_k \leftarrow [b_{01} \ldots b_{ij} \ldots b_{k-2,k-1}] ) // ( C_k ) is a ( k \times \binom{k}{2} ) matrix</td>
</tr>
<tr>
<td>22</td>
<td>( D_k \leftarrow [A_k \mid C_k] ) // ( D_k ) is a ( k \times \binom{k+1}{2} ) matrix</td>
</tr>
<tr>
<td>23</td>
<td>( R \leftarrow D_k (A^2_{*k})^{-1} ) // ( R ) is a ( k \times \binom{k+1}{2} ) matrix</td>
</tr>
<tr>
<td>24</td>
<td>return ( R )</td>
</tr>
</tbody>
</table>
\texttt{Pgen} generates the $k \times \binom{k+1}{2}$ re-encryption matrix, $R$, as detailed in Algorithm 7.21. We construct $R$ so that it satisfies equation 7.1

$$R(a_{si} \circ a_{sj}) = g_{ij}p1 + \sum_{l=1}^{k-1} \sigma_{ijl}a_l$$

(7.1)

This ensures that, if $c$ is an encryption of $m$ and $c'$ is an encryption of $m'$, then $R(c \circ c') \pmod{pq}$ is a valid encryption of $mm'$. The reasoning for equation 7.1 is given in the proof of Theorem 10.

We also construct $R$ so that it satisfies the following identity:

\textbf{Lemma 8.} Let $A_{\times k} = [a_{0} \ a_{1} \ ... \ a_{k-1}]$, where the columns of $A_k$ form a basis for $\mathbb{Z}_{pq}^k$. If $RA_{\times k} = A_k$, then $Rv_{\times} = v$ for all $v \in \mathbb{Z}_{pq}^k$.

This guarantees that re-encrypting a ciphertext by applying $R$ reveals no new information.

\texttt{Pgen} returns a valid matrix $R$ provided the $A_{\times k}$ has an inverse. We prove that this is true with high probability in Theorem 9. In the unlikely event that this is not true, \texttt{Pgen} exits and we use \texttt{KGen} to generate new vectors $a_1, \ldots, a_{k-1}$ until it is.

\textbf{Theorem 9.} $A_{\times k}$ has no inverse \text{mod}_{pq} with probability at most $(k^2 - 1)(1/p + 1/q)$.

Note that Theorem 9 subsumes Lemma 7, since the first $k$ columns of $A_{\times k}$ contain $A_k$ as a submatrix, and must be linearly independent.

**Computational overhead**

The computational overhead increases, the number of arithmetic operations per plaintext multiplication is $O(k^3)$, and the space requirement per ciphertext is $O(k)$, by comparison with HE1.

**Encryption**

A plaintext, $m \in [0, M]$, is enciphered as $c$, a $k$-vector, using Algorithm 7.22.
7.4. GENERALISATION TO K DIMENSIONS

Algorithm 7.22: Enc: Encryption Algorithm

Input : \( m \in \mathcal{M} \)
Input : \((p, a_1, \ldots, a_{k-1})\): secret key
Input : modulus: public modulus
Output: \( c \in C \)

1. \( q \leftarrow \text{modulus}/p \)
2. \( r \leftarrow [0, q) \)
3. \textbf{foreach} \( j \in [1, k) \) \textbf{do}
   4. \( s_j \leftarrow [0, \text{modulus}) \)
5. \textbf{end}
6. \( c \leftarrow (m + rp)1 + \sum_{j=1}^{k-1} s_j a_j \pmod{\text{modulus}} \)
7. \textbf{return} \( c \)

Decryption.

Algorithm 7.23: Dec: Decryption Algorithm

Input : \( c \in C \)
Input : \((p, a_1, \ldots, a_{k-1})\): secret key
Output: \( m \in \mathcal{M} \)

1. \( A_k \leftarrow [a_0 \ a_1 \ldots a_{k-1}] \)
2. \( \gamma^T \leftarrow (A_k^{-1})_1 \) \hspace{1cm} // \((A_k^{-1})_1\) is the first row of \(A_k^{-1}\)
3. \( m \leftarrow \gamma^T c \pmod{p} \)
4. \textbf{return} \( m \)

A ciphertext is decrypted using Algorithm 7.23. We call \( \gamma \) the decryption vector, as in HE2.

Addition.

Addition is the vector sum of the ciphertext vectors as in HE2 (see section 7.3.1). The Add algorithm is identical to Algorithm 7.15 with the exception that the vectors involved are \(k\)-vectors rather than 2-vectors.

Multiplication

The multiplication algorithm Mult is given by Algorithm 7.24.
Algorithm 7.24: Mult: multiplication algorithm

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$c_s \leftarrow U_k c$</td>
</tr>
<tr>
<td>2</td>
<td>$c_s' \leftarrow U_k c'$</td>
</tr>
<tr>
<td>3</td>
<td>result $\leftarrow R(c_s \circ c'_s) \pmod{\text{modulus}}$</td>
</tr>
<tr>
<td>4</td>
<td>return result</td>
</tr>
</tbody>
</table>

Theorem 10 states that Mult always returns a valid encryption of the product of the plaintexts of which $c$ and $c'$ are encryptions.

**Theorem 10.** If $c$ is an encryption of $m$ and $c'$ is an encryption of $m'$ then $R(c_s \circ c'_s) \pmod{pq}$ is an encryption of $mm'$.

**Security**

We construct $R$ so that it satisfies Equation 7.2:

$$RA^0_{sk} = [A_k \mid C_k].$$

(7.2)

which gives $k\binom{k+1}{2}$ linear equations for the $k\binom{k+1}{2}$ elements of $R$ in terms of quadratic functions of the $k(k-1)$ $a_{ij}$’s ($1 \leq i \leq k, 1 \leq j \leq k - 1$), which are undetermined. Thus the system has $k(k-1)$ parameters that cannot be deduced from $R$.

Each $c$ introduces $k$ new parameters $r, s_1, \ldots, s_{k-1}$ and $k$ equations, so the number of undetermined parameters is always $k(k-1)$.

Finally, as with HE1 and HE2, HE$k$ is vulnerable to CPA using the "collision" attack described in section 7.2.1.
Cryptanalysis

Note that \( p \) can be determined from \( m_i \) for \( k \) ciphertexts. Let

\[
C = [c_1 - m_1 1 \ldots c_k - m_k 1], \quad A_k = [1 a_1 \ldots a_{k-1}]
\]

and let

\[
W = \begin{bmatrix}
    r_1 p & r_2 p & \ldots & r_k p \\
    s_{1,1} & s_{2,1} & \ldots & s_{k,1} \\
    \vdots  & \vdots  & \ddots & \vdots  \\
    s_{1,k-1} & s_{2,k-1} & \ldots & s_{k,k-1}
\end{bmatrix}, \quad W' = \begin{bmatrix}
    r_1 & r_2 & \ldots & r_k \\
    s_{1,1} & s_{2,1} & \ldots & s_{k,1} \\
    \vdots  & \vdots  & \ddots & \vdots  \\
    s_{1,k-1} & s_{2,k-1} & \ldots & s_{k,k-1}
\end{bmatrix},
\]

where \( r_i, s_{ij} \) refer to \( c_i \). Then \( C = A_k W \), and so \( \det C = \det A_k \det W \). Note that \( \det W = p \det W' \), so \( \det C \) is a multiple of \( p \). Now \( \det C \) can be determined in \( O(k^3) \) time and, if it is nonzero, \( p \) can be determined as \( \gcd(\det C, pq) \).

**Lemma 11.** \( \Pr(\det C = 0 \mod pq) \leq (2k - 1)(1/p + 1/q) \).

Once we have recovered \( p \), we can use the known \( m_i \) to determine the decryption vector \( \gamma \), by solving linear equations. Let \( C_0 = [c_1 \ c_2 \ldots \ c_k], \ m^T = [m_1 \ m_2 \ldots \ m_k] \).

**Lemma 12.** \( \Pr(\det C_0 = 0 \mod pq) \leq (2k - 1)(1/p + 1/q) \).

Thus, with high probability, we can uniquely solve the system \( \gamma^T C_0 = m^T \mod p \), to recover \( \gamma \) and enable decryption of an arbitrary ciphertext. However, encryption of messages is not possible, since we gain little information about \( a_1, \ldots, a_k \). Note also that, if we determined \( p \) by some means other than using \( k \) known plaintexts, it is not clear how to recover \( \gamma \).

To break this system, we need to guess \( k \) plaintexts. The entropy of a \( k \)-tuple of plaintexts \( (m_1, m_2, \ldots, m_k) \) is \( kp \), so effectively we need \( \mu^k \) guesses, where \( \mu \) is the number of guesses needed to break HE1. So HE\( k \) can tolerate much smaller entropy than HE1, provided \( k \) is large enough. If \( k \) is sufficiently large, the scheme appears secure without adding noise, but does not have the other advantages of adding noise.
Fixing an insecurity for $k > 2$

The decryption vector for $HE_k$ is $\gamma^T = (A_k^{-1})_1$. Note that $\gamma^T 1 = 1$ and $\gamma^T a_i = 0$ ($i \in [1, k - 1]$), since $\gamma^T a_i = I_{1i}$ ($i \in [0, k - 1]$).

The equations

$$\gamma^T ((c_1 \cdot c_2) \cdot c_3) = \gamma^T (c_1 \cdot (c_2 \cdot c_3)) \pmod{pq},$$

(7.4)

in order to have correct decryption. The associator for $A_k$ is

$$[c_i, c_j, c_l] = c_i \cdot (c_j \cdot c_l) - (c_i \cdot c_j) \cdot c_l = rp1 + \sum_{i=1}^{k-1} s_ic_i \pmod{pq}.$$  

(7.5)

By linearity, (7.5) holds if and only if it holds for all basis elements, excluding the identity. That is, for all $i, j, l \in [1, k - 1]$, we need

$$\gamma^T ((a_i \cdot a_j) \cdot a_l) = \gamma^T ((a_i \cdot a_j) \cdot a_l) \pmod{q}.$$  

(7.6)
Algorithm 7.25: Pgen: Parameter Generation Algorithm (Revised)

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( q \leftarrow \text{modulus} / p )</td>
</tr>
<tr>
<td>2</td>
<td>( a_0 \leftarrow 1 )</td>
</tr>
<tr>
<td>3</td>
<td>( A_k \leftarrow [a_0, a_1, \ldots, a_{k-1}] )</td>
</tr>
<tr>
<td>4</td>
<td>( \text{foreach } j \in [0, k) ) ( \rightarrow ) ( a_{*j} \leftarrow U_k a_j )</td>
</tr>
<tr>
<td>5</td>
<td>( \text{foreach } i, j \in [0, k) ), ( 0 \leq i \leq j &lt; k ) ( \rightarrow ) ( a_{{*ij}}^2 \leftarrow a_{*i} \odot a_{*j} )</td>
</tr>
<tr>
<td>6</td>
<td>( A_{*k}^{\odot 2} \leftarrow [a_{{*ij}}^{\odot 2} \ldots a_{{*k-2,k-1}}^{\odot 2}] ) ( // A_{*k}^{\odot 2} ) ( ) is a ( \left( \frac{k+1}{2} \right) \times \left( \frac{k+1}{2} \right) ) matrix</td>
</tr>
<tr>
<td>7</td>
<td>( \text{if } A_{*k}^{\odot 2} ) has no inverse ( \rightarrow ) ( \text{return } \perp )</td>
</tr>
<tr>
<td>8</td>
<td>( \text{foreach } i, j \in [0, k), 0 \leq i \leq j &lt; k ) ( \rightarrow ) ( \varrho_i \overset{S}{\leftarrow} [0, q) )</td>
</tr>
<tr>
<td>9</td>
<td>( \varrho_i \overset{S}{\leftarrow} [0, q) )</td>
</tr>
<tr>
<td>10</td>
<td>( \varrho_{ij} \leftarrow \varrho_i \varrho_j \mod q )</td>
</tr>
<tr>
<td>11</td>
<td>( \text{repeat} )</td>
</tr>
<tr>
<td>12</td>
<td>( \tau \overset{S}{\leftarrow} [0, q) )</td>
</tr>
<tr>
<td>13</td>
<td>( \text{foreach } l \in [1, k) ) ( \rightarrow ) ( \sigma'_{ijl} \overset{S}{\leftarrow} [0, q) )</td>
</tr>
<tr>
<td>14</td>
<td>( \text{until } \sum_{l=1}^{k-1} \sigma'_{ijl} \varrho_l = \tau \varrho_i \varrho_j \mod q )</td>
</tr>
<tr>
<td>15</td>
<td>( \text{foreach } l \in [1, k) ) ( \rightarrow ) ( r_{ijl} \overset{S}{\leftarrow} [0, p) )</td>
</tr>
<tr>
<td>16</td>
<td>( \sigma_{ijl} \leftarrow \sigma'<em>{ijl} + r</em>{ijl} q )</td>
</tr>
<tr>
<td>17</td>
<td>( \text{end} )</td>
</tr>
<tr>
<td>18</td>
<td>( \varrho_{ij} \leftarrow \varrho_i \varrho_j \mod q )</td>
</tr>
<tr>
<td>19</td>
<td>( \text{end} )</td>
</tr>
<tr>
<td>20</td>
<td>( C_k \leftarrow [b_{01} \ldots b_{ij} \ldots b_{k-2,k-1}] ) ( // C_k ) ( ) is a ( k \times \left( \frac{k}{2} \right) ) matrix</td>
</tr>
<tr>
<td>21</td>
<td>( D_k \leftarrow [A_k \mid C_k] ) ( // D_k ) ( ) is a ( k \times \left( \frac{k+1}{2} \right) ) matrix</td>
</tr>
<tr>
<td>22</td>
<td>( R \leftarrow D_k(A_{*k}^{\odot 2})^{-1} ) ( // R ) ( ) is a ( k \times \left( \frac{k+1}{2} \right) ) matrix</td>
</tr>
<tr>
<td>23</td>
<td>( \text{return } R )</td>
</tr>
</tbody>
</table>
The associator for $A_k$ is

$$[a_i, a_j, a_l] = a_i \cdot (a_j \cdot a_l) - (a_i \cdot a_j) \cdot a_l = rp1 + \sum_{i=1}^{k-1} s_i a_l \pmod{pq},$$

for some integers $r, s_1, \ldots, s_{k-1}$, and so \( \gamma^T[a_i, a_j, a_l] = rp \).

If $A_k$ is associative, the problem does not arise, since (7.6) will be satisfied automatically. Associativity holds if $k \leq 2$. All we have to check is that $a \cdot (a \cdot a) = (a \cdot a) \cdot a$, which is true by commutativity. Thus HE1, HE2 cannot be attacked in this way. However, this collision attack becomes possible even when $k = 3$.

**Example** Suppose that $a_1 \cdot a_1 = a_2 \cdot a_2 = p1 + a_1 - a_2$, $a_1 \cdot a_2 = 2p1 + a_2$.

Then $a_1 \cdot (a_1 \cdot a_2) = a_1 \cdot (2p1 + a_2) = 2pa_1 + a_1 \cdot a_2 = 2p1 + 2pa_1 + a_2$, and $(a_1 \cdot a_1) \cdot a_2 = pa_2 + a_1 \cdot a_2 + a_2 \cdot a_2 = 3p1 + a_1 + pa_2$. So $[a_1, a_1, a_2] = p1 - (2p - 1)a_1 + (p + 1)a_2$, and $\gamma^T[a_1, a_1, a_2] = p$.

Requiring associativity in $A_k$ overconstrains the system, imposing $k(k+1)$ equations on the $k(k+1)$ structure constants. With only $k(k - 1)$ undetermined parameters, this is too much. But all we need is that (7.6) holds. We have

**Lemma 13.** Equation (7.6) holds if and only if $\sum_{i=1}^{k-1} \sigma_{ijl} q_{it} = \sum_{l=1}^{k-1} \sigma_{ijl} q_{it}$ (mod $q$), $\forall i, j, l \in [1, k - 1]$.

Now we can ensure (7.6) by giving the $q_{ij}$ a multiplicative structure.

**Lemma 14.** Let $\tau, q_i \smile [0, q)$ ($i \in [1, k - 1]$), let $q_{ij} = q_i q_j$ mod $q$, and let the $\sigma_{ijl}$ satisfy $\sum_{i=1}^{k-1} \sigma_{ijl} q_t = \tau q_i q_j$ (mod $q$) for all $i, j \in [1, k - 1]$. Then, for all $i, j, l \in [1, k - 1]$, $\gamma^T(a_i \cdot (a_j \cdot a_l)) = \tau q_i q_j q_l$ mod $q$, the symmetry of which implies (7.6).

Thus the conditions of Lemma 14 are sufficient to remove the insecurity. The price is that we now have $(k - 1)(\binom{k}{2}) + (k - 1) + k(k - 1) = (k + 1)(\binom{k}{2}) + k - 1$ parameters and $k(\binom{k}{2})$ equations. There are $\binom{k}{2} + (k - 1) = (k + 2)(k - 1)/2$ independent parameters. This is fewer than the original $k(k - 1)$, but remains $\Omega(k^2)$.

As a result, the parameter generation algorithm $P\text{gen}$ is amended to Algorithm 7.25.
7.4.2 Insufficient entropy (HE$\kappa N$)

We generalise HE2N to $k$ dimensions.

Key generation.

**Algorithm 7.26: KGen: Key Generation Algorithm**

<table>
<thead>
<tr>
<th>Input</th>
<th>$\lambda \in S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>$\rho \in \mathbb{Z}$: entropy of inputs</td>
</tr>
<tr>
<td>Input</td>
<td>$\rho' \in \mathbb{Z}$: entropy of inputs</td>
</tr>
<tr>
<td>Output:</td>
<td>$(p, a_1, \ldots, a_{k-1})$: secret key</td>
</tr>
<tr>
<td>Output:</td>
<td>modulus: public modulus</td>
</tr>
</tbody>
</table>

1. $\nu \leftarrow \rho' - \rho$
2. $\kappa \leftarrow \rho' \mod [2^{\nu-1}, 2^\nu]
3. $p \leftarrow \rho' \mod [2^{\lambda-1}, 2^\lambda]
4. $\eta \leftarrow \lambda^2 / \rho - \lambda$
5. $q \leftarrow \rho' \mod [2^{\nu-1}, 2^\nu]
6. modulus $\leftarrow pq$
7. $a_0 \leftarrow 1$
8. repeat
9. $a_j \leftarrow [1, \modulus]^k, j \in [1, k)$
10. until $a_j, j \in [0, k)$, form a basis for $\mathbb{Z}^k_{pq}$
11. return $(p, a_1, \ldots, a_{k-1}), \text{modulus}$

KGen (Algorithm 7.26) randomly chooses $\kappa$, $p$ and $q$ according to the bounds outlined in section 7.3.2 and sets $a_j \forall j$. Note that, as a result of adding the “noise” term, defence against non-associativity is not required. Therefore, $Pgen$, which generates $R$, is Algorithm 7.21. The secret key, $sk$, is $(\kappa, p, a_1, \ldots, a_{k-1})$, and the public parameters are $pq$ and $R$.

Encryption.

A plaintext, $m \in [0, M]$, is enciphered by Algorithm 7.27.

Decryption.

A ciphertext is deciphered by Algorithm 7.28.
**Algorithm 7.27**: Enc: Encryption Algorithm

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( q \leftarrow ) modulus/p</td>
</tr>
<tr>
<td>2</td>
<td>( r \leftarrow ) ([0, q])</td>
</tr>
<tr>
<td>3</td>
<td>( s \leftarrow ) ([0, \kappa])</td>
</tr>
<tr>
<td>4</td>
<td>( \text{foreach } j \in [1, k] \text{ do} )</td>
</tr>
<tr>
<td>5</td>
<td>( t_j \leftarrow ) ([0, \text{modulus}])</td>
</tr>
<tr>
<td>6</td>
<td>( \text{end} )</td>
</tr>
<tr>
<td>7</td>
<td>( c \leftarrow (m + rp + sk)1 + \sum_{j=1}^{k-1} t_j a_j \text{ (mod modulus)} )</td>
</tr>
<tr>
<td>8</td>
<td>return ( c )</td>
</tr>
</tbody>
</table>

**Algorithm 7.28**: Dec: Decryption Algorithm

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( A_k \leftarrow [a_0 a_1 \ldots a_{k-1}] )</td>
</tr>
<tr>
<td>2</td>
<td>( \gamma^T \leftarrow (A_k^{-1})_1 ) \hspace{1cm} // (A_k^{-1})_1 \text{ is the first row of } A_k^{-1}</td>
</tr>
<tr>
<td>3</td>
<td>( m \leftarrow (\gamma^T c \mod p) \mod \kappa )</td>
</tr>
<tr>
<td>4</td>
<td>return ( m )</td>
</tr>
</tbody>
</table>
7.5. EXTENDING HE2N USING THE CRT

Arithmetic.

The addition and multiplication algorithms are as in section 7.4.1.

Security.

The effective entropy of HE$k$N is $\rho' = k(\rho + \log_2 \kappa)$. Thus, as we increase $k$, the “noise” term can be made smaller while still providing the requisite level of entropy.

Clearly HE$k$N also inherits the conclusions of Theorem 3, so this system also satisfies IND-CPA.

7.5 An extension of HE2N using the Chinese Remainder Theorem (HE2NCRT)

As an interesting aside, we extend HE2N (section 7.3.2) using a technique inspired by Chinese Remainder Theorem (CRT) secret sharing, so that we compute the final result modulo a product of primes $\prod_{j=1}^{K} p_j$ rather than modulo $p$, where $K$ is the number of primes.

In this scheme, we distribute the computation. We have $K$ sets of processors. Each processor computes arithmetic on ciphertexts modulo $p_jq_j$, where $p_j, q_j$ are suitable primes. Also, each processor only receives the $j$th ciphertext vector of an integer. Addition and multiplication of ciphertexts is as defined in section 7.3.1, except that it is performed modulo $p_jq_j$.

This serves two purposes. The first is to be able to handle larger moduli by dividing the computation into subcomputations on smaller moduli. The second is to mitigate against exposure of the secret key $p$ in the system presented in section 7.3.2 by not distributing the modulus $pq$ to each processor. Instead, we distribute $p_jq_j$ to the $j$th processor, for $j \in [1, K]$. This allows us to partition the computation into subcomputations, encrypted using different parameters. Thus, should an attacker compromise one subcomputation, they may gain no knowledge of other subcomputations. With regard to MapReduce, this scheme could easily
be applied to the map only jobs.

### Key Generation

<table>
<thead>
<tr>
<th><strong>Algorithm 7.29:</strong> KGen: Key Generation Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong></td>
</tr>
<tr>
<td><strong>Input:</strong></td>
</tr>
<tr>
<td><strong>Input:</strong></td>
</tr>
<tr>
<td><strong>Output:</strong></td>
</tr>
<tr>
<td><strong>Output:</strong></td>
</tr>
</tbody>
</table>

1. $\nu \leftarrow \rho' - \rho$
2. $\kappa \leftarrow \frac{\nu}{2^{\nu - 1}, 2^\nu}$
3. $\eta \leftarrow \frac{\lambda^2}{\rho'} - \lambda$
4. **foreach** $j \in [1, K]$ **do**
5. \[ p_j \leftarrow \frac{2^{\lambda - 1}, 2^\lambda}{p} \]
6. \[ q_j \leftarrow \frac{2^{\eta - 1}, 2^\eta}{p} \]
7. $\text{modulus}_j \leftarrow p_j q_j$
8. **repeat**
9. \[ a_{j,i} \leftarrow [1, \text{modulus}_j] (i \in [1, 2]) \]
10. **until** $a_{j,1}, a_{j,2}, a_{j,1} - a_{j,2} \neq 0 \pmod{p \text{ and } q}$
11. \[ a_j \leftarrow [a_{j,1}, a_{j,2}]^T \]
12. **end**
13. **return** $(\kappa, p_1, p_2, \ldots, p_K, a_1, \ldots, a_K), \text{modulus}_1, \text{modulus}_2, \ldots, \text{modulus}_K$

The key generation process, KGen (Algorithm 7.29), randomly chooses $\kappa$ as in section 7.2.2. For all $j \in [1, K]$, it randomly chooses a prime $p_j$ such that $p_j$ satisfies $2^{\lambda - 1} < p_j < 2^\lambda$ and

$$\Pi = \prod_{j=1}^{K} p_j > (n + 1)^d (M + \kappa^2)^d$$

It also randomly chooses $q_j, j \in [1, K]$, as for $q$ in section 7.2.1. Finally, it sets $a_j = [a_{j,1} \ a_{j,2}]^T$, where $a_{j,k} \leftarrow [1, p_j q_j]$ ($j \in [1, K], k \in [1, 2]$) such that $a_{j,1} \neq a_{j,2} \pmod{p}$ and $a_{j,1} \neq a_{j,2} \pmod{q}$.

The parameter generation algorithm Pgen (Algorithm 7.30) generates each re-encyption matrix $R_j (j \in [1, K])$.

The secret key, sk, is $(\kappa, p_1, \ldots, p_K, a_1, \ldots, a_K)$, and the public parameters are
7.5. EXTENDING HE2N USING THE CRT

Algorithm 7.30: Pgen: Parameter Generation Algorithm

\textbf{Input :} \((\kappa, p_1, p_2, \ldots, p_K, a_1, \ldots, a_K)\): secret key
\textbf{Input :}\ modulus_1, modulus_2, \ldots, modulus_K: public moduli
\textbf{Output :} \(R_1, R_2, \ldots, R_K\): public re-encryption matrices

\begin{enumerate}
\item \textbf{foreach} \(j \in [1, K]\) \textbf{do}
\item \(q_j \leftarrow \text{modulus}_j/p_j\)
\item \(\beta_j \leftarrow 2(a_j - a_{j1})^2\)
\item \(\varrho_j \leftarrow [0, q_j]\)
\item \(\beta_j^{-1}(\sigma_j a_{j1} + \varrho_j p - a_{j1}^2)\)
\item \(\alpha_{j1} \leftarrow \beta_j^{-1}(\sigma_j a_{j1} + \varrho_j p - a_{j1}^2)\)
\item \(\alpha_{j2} \leftarrow \beta_j^{-1}(\sigma_j a_{j2} + \varrho_j p - a_{j2}^2)\)
\item \(R_j \leftarrow \begin{bmatrix} 1 - 2\alpha_{j1} & \alpha_{j1} & \alpha_{j1} \\ -2\alpha_{j2} & \alpha_{j2} + 1 & \alpha_{j2} \end{bmatrix}\)
\item \textbf{end}
\item \textbf{return} \(R_1, \ldots, R_K\)
\end{enumerate}

\(p_j q_j\) (\(j \in [1, K]\)) and \(R_j\) (\(j \in [1, K]\)).

Encryption

Algorithm 7.31: Enc: Encryption Algorithm

\textbf{Input :} \(m \in M\)
\textbf{Input :} \((\kappa, p_1, p_2, \ldots, p_K, a_1, \ldots, a_K)\): secret key
\textbf{Input :}\ modulus_1, modulus_2, \ldots, modulus_K: public moduli
\textbf{Output :} \(c_1, c_2, \ldots, c_K \in C\)

\begin{enumerate}
\item \textbf{foreach} \(j \in [1, K]\) \textbf{do}
\item \(q_j \leftarrow \text{modulus}_j/p_j\)
\item \(r_j \leftarrow [0, q_j]\)
\item \(s_j \leftarrow [0, \kappa]\)
\item \(t_j \leftarrow [0, \text{modulus}_j]\)
\item \(c_j \leftarrow (m + r_j p_j + s_j \kappa) 1 + t_j a_j \mod \text{modulus}_j\)
\item \textbf{end}
\item \textbf{return} \(c_1, c_2, \ldots, c_K\)
\end{enumerate}

We encrypt an integer, \(m \in M\), as the set of \(K\) 2-vectors, \(c_j\) using Algorithm 7.31.
Decryption

Algorithm 7.32: Dec: Decryption Algorithm

\begin{algorithm}
\begin{algorithmic}
\STATE \textbf{Input} : $c_1, c_2, \ldots, c_K \in C$
\STATE \textbf{Input} : $(\kappa, p_1, p_2, \ldots, p_K, a_1, \ldots, a_K)$: secret key
\STATE \textbf{Output}: $m \in M$
\STATE $\Pi \leftarrow \prod_{j=1}^{K} p_j$
\FOR{\textbf{foreach} $j \in [1, K]$}
\STATE $\gamma_j^T \leftarrow (a_{j2} - a_{j1})^{-1}[a_{j2} - a_{j1}]$
\STATE $d_j \leftarrow (\gamma_j^T c_j \mod p_j)$
\STATE $M_j \leftarrow \Pi / p_j$
\STATE $\mu_j \leftarrow M_j^{-1} \mod p_j$
\ENDFOR
\STATE $m \leftarrow \left( \sum_{j=1}^{K} d_j M_j \mu_j \mod \Pi \right) \mod \kappa$
\STATE \textbf{return} $m$
\end{algorithmic}
\end{algorithm}

To decrypt a set of $K$ ciphertexts we use Algorithm 7.32. We first decrypt the $j$th ciphertext of the computational result, $c_j$, as in section 7.3.1, to give

$$P_j = (a_{j2} - a_{j1})^{-1}(a_{j2}c_{j1} - a_{j1}c_{j2}) \mod p_j,$$

where $P_j$ is the residue of $P(m_1, m_2, \ldots, m_n, \kappa) \mod p_j$.

We then use the Chinese Remainder Theorem to compute the plaintext as

$$P(m_1, m_2, \ldots, m_n) = \left( \sum_{j=1}^{K} P_j M_j \mu_j \mod \Pi \right) \mod \kappa,$$

where $M_j = \Pi / p_j$ and $\mu_j = M_j^{-1} \mod p_j$.

Arithmetic

Addition of ciphertexts on processor $j$ is performed using Algorithm 7.33. Multiplication of ciphertexts on processor $j$ is performed by Algorithm 7.34.
7.6. **FULLY HOMOMORPHIC SYSTEM**

We return to HE\(k\), presented above in section 7.4.1. We will show that, for large enough \(k\), this can be made into a fully homomorphic system.

Suppose we have inputs \(m_1, m_2, \ldots, m_n\), where \(m_i \in [0, M]\), \(i \in [1, n]\), \(p, q\) are large primes, and we want to evaluate an arithmetic circuit, \(\Phi\), over the ring \(\mathbb{Z}_{pq}\), on these inputs.

From [165], the definition of an arithmetic circuit is:

**Definition 22** (Arithmetic Circuits.). An arithmetic circuit \(\Phi\) over the ring \(\mathbb{Z}_{pq}\).

---

**Algorithm 7.33:** Add\(_j\): addition algorithm for processor \(j\)

<table>
<thead>
<tr>
<th>Input</th>
<th>(c_j \in \mathcal{C})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>(c'_j \in \mathcal{C})</td>
</tr>
<tr>
<td>Input</td>
<td>modulus(_j \in \mathbb{Z}): modulus for arithmetic</td>
</tr>
<tr>
<td>Output</td>
<td>result(_j \in \mathcal{C})</td>
</tr>
</tbody>
</table>

1. \(\text{result}_j \leftarrow c_j + c'_j \pmod{\text{modulus}_j}\)
2. return \(\text{result}_j\)

---

**Algorithm 7.34:** Mult\(_j\): multiplication algorithm for processor \(j\)

<table>
<thead>
<tr>
<th>Input</th>
<th>(c_j = [c_{j1} c_{j2}]^T \in \mathcal{C})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>(c'<em>j = [c'</em>{j1} c'_{j2}]^T \in \mathcal{C})</td>
</tr>
<tr>
<td>Input</td>
<td>modulus(_j \in \mathbb{Z}): public modulus</td>
</tr>
<tr>
<td>Input</td>
<td>(R_j): re-encryption matrix</td>
</tr>
<tr>
<td>Output</td>
<td>result(_j \in \mathcal{C})</td>
</tr>
</tbody>
</table>

1. \(c_{j3} \leftarrow 2c_{j1} - c_{j2}\)
2. \(c_{j*} \leftarrow [c_{j1} c_{j2} c_{j3}]^T\)
3. \(c'_{j3} \leftarrow 2c'_{j1} - c'_{j2}\)
4. \(c'_{j*} \leftarrow [c'_{j1} c'_{j2} c'_{j3}]^T\)
5. \(\text{result}_j \leftarrow R_j(c_{j*} \circ c'_{j*}) \pmod{\text{modulus}_j}\)
6. return \(\text{result}_j\)

---

**Extending HE2NCRT to \(k\)-vectors**

Clearly HE\(k\)\(N\) could be extended to HE\(k\)\(NCRT\) in a similar way to the extension of HE2 to HE\(k\). Future work will discuss the details of such a scheme.
and the variables $X = \{x_1, \ldots, x_n\}$ is a directed acyclic graph with every node of in-degree either two or zero, labelled in the following manner: every vertex of in-degree 0 is labelled by either a variable in $X$ or an element of $\mathbb{R}$. Every other node in $\Phi$ has in-degree two and is labelled by either $\times$ or $\pm$. A circuit $\Phi$ computes a polynomial $f \in \mathbb{R}[X]$ in the obvious manner. An arithmetic circuit is called a formula if the out-degree of each node in it is one (and so the underlying graph is a directed tree). The size of a circuit is the number of nodes in it, and the depth of a circuit is the length of the longest directed path in it.

To transform HE$k$ to a fully homomorphic system, we define encryption of the circuit inputs as in HE$k$. Similarly, addition and multiplication of ciphertexts at each arithmetic node of the circuit is defined as in the HE$k$ scheme. This way we are able to compute the arithmetic circuit homomorphically. However, this system is still “somewhat” homomorphic. If the computational result grows larger than $p$, we are unable to successfully decrypt the result. This restricts us to circuits of bounded depth to avoid this plaintext blow up. To make it fully homomorphic, we consider Boolean circuits \[331\]. A Boolean circuit is defined in \[331\] as:

**Definition 23** (Boolean circuit.). We define a basis of Boolean functions, $B$, where each member of $B$ is a function, $f : [0,1]^m \rightarrow [0,1]$, for some $m \in \mathbb{N}$. Therefore, a Boolean circuit over a basis $B$, with $n$ inputs and $m$ outputs, is then defined as a finite directed acyclic graph. Each vertex corresponds to either a basis function, which we call a gate, or one of the inputs, and there are a set of exactly $m$ nodes which are labeled as the outputs. The edges must also have some ordering, to distinguish between different arguments to the same Boolean function.

We note that any finite computation can be represented as a Boolean circuit.

Typically, the basis $B$ will be the Boolean functions, AND, OR, and NOT. However, a Boolean circuit may be alternatively represented using only NAND gates \[293\]. The indegree of any gate in the directed acyclic graph $G = (V,E)$ is then always 2, but the outdegree may be arbitrary. We will refer to the directed edges of this graph as wires. We note that, for a Boolean circuit, the inputs to each gate are bits, as are the outputs. We will denote the set of inputs to the circuit by $I \subseteq V$, and the set of outputs from the circuit by $O \subseteq V$. In $G$, the inputs
7.6. **FULLY HOMOMORPHIC SYSTEM**

\[ x \in \{\alpha_0, \alpha_1\} \]

\[ L(x, y) \]

\[ w \in \{\gamma_0, \gamma_1\} \]

Figure 7.1: A wire from an input

have indegree 0, and the outputs have outdegree 0, but we will regard the inputs as having indegree 1, and the outputs as having outdegree 1, with wires from and to the external environment \( \Lambda \).

Note that, if we represent the bit values 0, 1 with known constants \( \alpha_0, \alpha_1 \), the system is open to attack, even though the inputs are encrypted with HE\( k \). For any ciphertext \( c \), we can take

\[ c' = \begin{cases} 
  c - \alpha_0 1, & \text{with probability } \frac{1}{2}, \\
  c - \alpha_1 1, & \text{with probability } \frac{1}{2}.
\end{cases} \]

Then \( c' \) is an encryption of 0 with probability \( \frac{1}{2} \). If we multiply \( \nu \) of these ciphertexts, the probability that none of them is an encryption of zero is \( 2^{-\nu} \), so we obtain an encryption of zero with probability \( 1 - 2^{-\nu} \). So, with \( \nu \) repetitions of this process, if \( \nu = 2 \lg k \), this is \( 1 - 1/k \). By repeating the whole process \( \Omega(k) \) times, we can obtain \( k \) encryptions of zero with high probability. Once we have done this, we can use the gcd method of [4] to determine \( p \).

Therefore, we encrypt both the inputs to the circuit, and the values on the wires, using HE\( k \). On each wire \( e \in E \), we will represent the bit value \( b_e \in \{0, 1\} \) by \( w_e \in \{\omega_{0e}, \omega_{1e}\} \), where \( \omega_{0e}, \omega_{1e} \in [0, q) \) are random even and odd integers respectively. Thus \( b_e = w_e \mod 2 \). For each input \( i \in I \), we represent the input bit value \( b_i \) similarly, by \( x_i \in \{\omega_{0i}, \omega_{1i}\} \).

An input \( i \in I \) has a wire \((\Lambda, i)\) on which the (encrypted) input value \( x_i \) is stored. For any wire \( e = (i, v) \) from input \( i \), we have a linear function \( L(x) = a + bx \), which converts the plaintext input value \( x \in \{\alpha_0, \alpha_1\} \) to the wire value \( w \in \{\gamma_0, \gamma_1\} \). It
is easy to check that this requires

\[ a = (\alpha_1 - \alpha_0)^{-1}(\alpha_1 \gamma_0 - \alpha_0 \gamma_1), \quad b = (\alpha_1 - \alpha_0)^{-1}(\gamma_1 - \gamma_0). \]

The encrypted coefficients of this function are stored as data for the wire \( e \) (see Fig. 7.1). Note that all computations are \( \mod pq \), and the required inverses exist because the numbers involved are less than \( q \).

For each output wire \( e = (v, v') \) of a NAND gate \( v \), we have a quadratic function \( Q(x, y) = a + bx + cy + dxy \), which converts the values on the input wires of the gate, \( x \in \{\alpha_0, \alpha_1\} \), \( y \in \{\beta_0, \beta_1\} \), to the wire value \( w \in \{\gamma_0, \gamma_1\} \). It is easy to check that this requires

\[ a = \gamma_0 + \alpha_1 \beta_1 \vartheta, \quad b = -\beta_1 \vartheta, \quad c = -\alpha_1 \vartheta, \quad d = \vartheta, \]

where \( \vartheta = ((\alpha_1 - \alpha_0)(\beta_1 - \beta_0))^{-1}(\gamma_1 - \gamma_0) \). Again, the encrypted coefficients of this function are stored as data for the wire \( e \) (see Fig. 7.2).

For each output NAND gate \( v \in O \), we decrypt the value \( w \in \{\gamma_0, \gamma_1\} \) computed by its (unique) output wire \( (v, \Lambda) \). Then the output bit is \( w \mod 2 \).

Thus we have replaced the logical operations of the Boolean circuit by evaluation of low degree polynomials. For simplicity, we have chosen to use only NAND gates, but we can represent any binary Boolean function by a quadratic polynomial in the way described above. Since the quadratic polynomials are encrypted in our system, they conceal the binary Boolean function they represent. Thus the circuit can be “garbled”\[30\][145], to minimise inference about the inputs and outputs of the circuit from its structure.
However, there is a price to be paid for controlling the noise. The encrypted circuit is not securely reusable with the same values $\omega_0^e, \omega_1^e$ for $w_e$. Suppose we can observe the encrypted value on wire $e$ three times giving cyphertexts $c_1, c_2, c_3$. Two of these are encryptions of the same value $2s_{0,e}$ or $1 + 2s_{1,e}$. Thus $(c_1 - c_2) \cdot (c_1 - c_3) \cdot (c_2 - c_3)$ is an encryption of 0. By doing this for $k$ wires, we can break the system. This is essentially the collision attack described in section 7.2.

Some reuse of the encrypted circuit is possible by using multiple values on the wires, and higher degree polynomials for the gates. However, we will not consider this refinement, since the idea seems to have little practical interest.

As we discuss in section 7.4.1, the security of HE$k$ is exponential in $k$. So by setting $k$ large enough we make brute force attacks on the system infeasible. (A brute force attack succeeds in polynomial time only if $k = O(\log \log p).$) Therefore, if we compute encrypted Boolean circuits, our system is fully homomorphic.

7.7 Experimental Results

HE1, HE1N, HE2, and HE2N have been implemented in pure unoptimised Java using the JScience mathematics library [103]. Secure pseudo-random numbers are generated using the ISAAC algorithm [174], seeded using the Linux /dev/random source. This prevents the weakness in ISAAC shown by Aumasson [22].

The evaluation experiment generated 24,000 encrypted inputs and evaluated a polynomial homomorphically on the inputs, using a Hadoop MapReduce (MR) algorithm. On the secure client side, the MR input is generated as pseudo-random $\rho$-bit integers which are encrypted and written to a file with $d$ inputs per line, where $d$ is the degree of the polynomial to be computed. The security parameters $\lambda$ and $\eta$ were selected to be the minimum values required to satisfy the conditions give in sections 7.2.1, 7.2.2, 7.3.1, and 7.3.2. In addition, the unencrypted result of the computation is computed so that it may checked against the decrypted result of the homomorphic computation. On the Hadoop cluster side, each mapper processes a line of input by homomorphically multiplying together each input on a line and outputs this product. A single reducer homomorphically sums the products. The MR algorithm divides the input file so that each mapper receives an equal number of lines of input, ensuring maximum parallelisation. Finally, on
Table 7.1: Timings for each experimental configuration ($n = 24000$ in all cases, $\lambda > 96$). *Init* is the initialisation time for the encryption algorithm, *Enc* is the mean time to encrypt a single integer, *Exec* is the total MR job execution time, *Prod* is the mean time to homomorphically compute the product of two encrypted integers, *Sum* is the mean time to homomorphically compute the sum of two encrypted integers.

<table>
<thead>
<tr>
<th>Alg. Parameters</th>
<th>Encryption</th>
<th>MR Job</th>
<th>Decrypt</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d$  $\rho$ $\rho'$</td>
<td>Init(s)</td>
<td>Enc($\mu$s)</td>
<td>Exec(s)</td>
</tr>
<tr>
<td>HE1 2 32 n/a</td>
<td>0.12</td>
<td>13.52</td>
<td>23.82</td>
</tr>
<tr>
<td>HE1 2 64 n/a</td>
<td>0.12</td>
<td>16.24</td>
<td>23.85</td>
</tr>
<tr>
<td>HE1 2 128 n/a</td>
<td>0.15</td>
<td>25.73</td>
<td>23.77</td>
</tr>
<tr>
<td>HE1 3 32 n/a</td>
<td>0.17</td>
<td>22.98</td>
<td>23.65</td>
</tr>
<tr>
<td>HE1 3 64 n/a</td>
<td>0.19</td>
<td>34.63</td>
<td>24.72</td>
</tr>
<tr>
<td>HE1 3 128 n/a</td>
<td>0.42</td>
<td>54.83</td>
<td>26.05</td>
</tr>
<tr>
<td>HE1 4 32 n/a</td>
<td>0.28</td>
<td>43.36</td>
<td>24.48</td>
</tr>
<tr>
<td>HE1 4 64 n/a</td>
<td>0.53</td>
<td>58.85</td>
<td>26.41</td>
</tr>
<tr>
<td>HE1 4 128 n/a</td>
<td>1.36</td>
<td>104.95</td>
<td>28.33</td>
</tr>
<tr>
<td>HE2N 2 1 128</td>
<td>0.22</td>
<td>32.99</td>
<td>22.94</td>
</tr>
<tr>
<td>HE2N 2 1 64</td>
<td>0.39</td>
<td>52.63</td>
<td>26.24</td>
</tr>
<tr>
<td>HE2N 2 1 128</td>
<td>1.2</td>
<td>89.01</td>
<td>26.18</td>
</tr>
<tr>
<td>HE2N 2 8 32</td>
<td>0.6</td>
<td>57.88</td>
<td>25.9</td>
</tr>
<tr>
<td>HE2N 2 8 64</td>
<td>0.32</td>
<td>43.93</td>
<td>26.53</td>
</tr>
<tr>
<td>HE2N 2 8 128</td>
<td>1.13</td>
<td>78.11</td>
<td>24.42</td>
</tr>
<tr>
<td>HE2N 2 16 64</td>
<td>0.33</td>
<td>53.97</td>
<td>27.15</td>
</tr>
<tr>
<td>HE2N 2 16 128</td>
<td>0.63</td>
<td>68.73</td>
<td>25.22</td>
</tr>
<tr>
<td>HE2N 3 1 32</td>
<td>8.54</td>
<td>183.19</td>
<td>24.24</td>
</tr>
<tr>
<td>HE2N 3 1 64</td>
<td>3.67</td>
<td>125</td>
<td>29.49</td>
</tr>
<tr>
<td>HE2N 3 1 128</td>
<td>27.84</td>
<td>313.76</td>
<td>26.94</td>
</tr>
<tr>
<td>HE2N 3 8 32</td>
<td>115</td>
<td>462.45</td>
<td>32.61</td>
</tr>
<tr>
<td>HE2N 3 8 64</td>
<td>9.75</td>
<td>180.08</td>
<td>25.87</td>
</tr>
<tr>
<td>HE2N 3 8 128</td>
<td>36.05</td>
<td>259.15</td>
<td>30.1</td>
</tr>
<tr>
<td>HE2N 3 16 64</td>
<td>30.96</td>
<td>378.99</td>
<td>28.24</td>
</tr>
<tr>
<td>HE2N 3 16 128</td>
<td>8.13</td>
<td>226.32</td>
<td>27.92</td>
</tr>
<tr>
<td>HE2N 2 32 n/a</td>
<td>0.16</td>
<td>85.79</td>
<td>26.82</td>
</tr>
<tr>
<td>HE2N 2 64 n/a</td>
<td>0.17</td>
<td>95.92</td>
<td>29.71</td>
</tr>
<tr>
<td>HE2N 2 128 n/a</td>
<td>0.22</td>
<td>132.53</td>
<td>32.84</td>
</tr>
<tr>
<td>HE2N 3 32 n/a</td>
<td>0.23</td>
<td>130.3</td>
<td>31.18</td>
</tr>
<tr>
<td>HE2N 3 64 n/a</td>
<td>0.29</td>
<td>145.62</td>
<td>32.84</td>
</tr>
<tr>
<td>HE2N 3 128 n/a</td>
<td>0.52</td>
<td>249.47</td>
<td>29.54</td>
</tr>
<tr>
<td>HE2N 4 32 n/a</td>
<td>0.39</td>
<td>175.63</td>
<td>29.5</td>
</tr>
<tr>
<td>HE2N 4 64 n/a</td>
<td>0.7</td>
<td>255.3</td>
<td>29.55</td>
</tr>
<tr>
<td>HE2N 4 128 n/a</td>
<td>2.7</td>
<td>465.51</td>
<td>37.47</td>
</tr>
<tr>
<td>HE2N 2 1 32</td>
<td>0.27</td>
<td>147.83</td>
<td>29.74</td>
</tr>
<tr>
<td>HE2N 2 1 64</td>
<td>0.44</td>
<td>202.74</td>
<td>33.36</td>
</tr>
<tr>
<td>HE2N 2 1 128</td>
<td>1.58</td>
<td>354.19</td>
<td>33.76</td>
</tr>
<tr>
<td>HE2N 2 8 32</td>
<td>0.59</td>
<td>234.84</td>
<td>31.42</td>
</tr>
<tr>
<td>HE2N 2 8 64</td>
<td>0.33</td>
<td>163.78</td>
<td>27.42</td>
</tr>
<tr>
<td>HE2N 2 8 128</td>
<td>0.9</td>
<td>307.68</td>
<td>36.32</td>
</tr>
<tr>
<td>HE2N 2 16 64</td>
<td>0.42</td>
<td>208.1</td>
<td>29.96</td>
</tr>
<tr>
<td>HE2N 2 16 128</td>
<td>0.73</td>
<td>274.48</td>
<td>30.82</td>
</tr>
<tr>
<td>HE2N 3 1 32</td>
<td>5.72</td>
<td>651.1</td>
<td>36.49</td>
</tr>
<tr>
<td>HE2N 3 1 64</td>
<td>4.45</td>
<td>477.52</td>
<td>35.33</td>
</tr>
<tr>
<td>HE2N 3 1 128</td>
<td>26.83</td>
<td>1102.79</td>
<td>43.23</td>
</tr>
<tr>
<td>HE2N 3 8 32</td>
<td>87.38</td>
<td>1658.36</td>
<td>49.63</td>
</tr>
<tr>
<td>HE2N 3 8 64</td>
<td>5.21</td>
<td>607.75</td>
<td>36.54</td>
</tr>
<tr>
<td>HE2N 3 8 128</td>
<td>17.14</td>
<td>945.64</td>
<td>40.49</td>
</tr>
<tr>
<td>HE2N 3 16 64</td>
<td>39.19</td>
<td>1368.18</td>
<td>44.88</td>
</tr>
<tr>
<td>HE2N 3 16 128</td>
<td>11.39</td>
<td>774.07</td>
<td>36.05</td>
</tr>
</tbody>
</table>
the secure client side, the MR output is decrypted.

Our test environment consisted of a single secure client (an Ubuntu Linux VM with 16GB RAM) and a Hadoop 2.7.3 cluster running in a heterogeneous OpenNebula cloud. The Hadoop cluster consisted of 17 Linux VMs, one master and 16 slaves, each allocated 2GB of RAM. Each experimental configuration of algorithm, polynomial degree \(d\), integer size \(\rho\), and effective entropy of inputs after adding “noise” \(\rho'\), for the ‘N’ variant algorithms only, was executed 10 times. The means are tabulated in Table 7.1.

There are some small anomalies in our data. This may be attributed to the random data set generation. Smaller integers may have been generated for particular test cases, resulting in faster execution times. In addition, JScience implements arbitrary precision integers as an array of Java \texttt{long} (64-bit) integers. This underlying representation may be optimal in some of our test configurations and suboptimal in others, causing anomalous results. Another possibility is that the unexpected results are due to garbage collection in the JVM heap, which may be more prevalent in certain test configurations. With regard to the algorithm initialisation times dominating in terms of execution time, being seconds rather than milliseconds, this can be explained by the fact that each HE algorithm generates large probable primes as secret and public parameters. This step in the initialisation process represents a significant overhead, particularly if the prime generation algorithm requires several iterations to generate a candidate prime. Finally, regarding MR job running times remaining roughly similar regardless of test case parameters, we attribute this to the overhead of the MR framework dominating the execution time. We surmise that the inter-process communication and creation and destruction of Java objects performed by the Hadoop framework is significant in comparison to the execution time of our algorithms. Indeed, we observed a similar phenomenon regarding the MR running times with our experiments recorded in chapter 8.

We may compare these results with those reported in the literature. Our results compare extremely favourably with Table 2 of [199]. For encryption, our results are, in the best case, 1000 times faster than those presented there, and, in the worst case, 10 times faster. For decryption, our results are comparable. However, it should be noted that to decrypt our results we take the moduli for large primes rather than 2 as in [199], which is obviously less efficient. For homomorphic sums
and products, our algorithms perform approximately 100 times faster. \cite{199} only provides experimental data for computing degree 2 polynomials. We provide experimental results for higher degree polynomials. Similarly, compared with Fig. 13 of Popa et al. \cite{272}, our encryption times for a 32-bit integer are considerably faster. While a time for computing a homomorphic sum on a column is given in Fig. 12, it is unclear how many rows exist in their test database. Nevertheless, our results for computing homomorphic sums compare favourably with those given. Since CryptDB \cite{272} only supports homomorphic sums and cannot compute an inner product, we can only compare the homomorphic sum timings.

Table 1 of \cite{311} is unclear whether the timings are aggregate or per operation. Even assuming that they are aggregate, our results are approximately 100 times faster for homomorphic sum and product operations. Crypsis \cite{311} uses two different encryption schemes for integers, ElGamal \cite{127} and Paillier \cite{262}, which only support addition or multiplication but not both. No discussion of computation of an inner product is made in \cite{311} but we expect that the timings would be considerably worse as data encrypted using ElGamal to compute the products would have to be shipped back to the secure client to be re-encrypted using Paillier so that the final inner product could be computed.

Varia et al. \cite{327} present experimental results of applying their HETest framework to HElib \cite{157}. Varia et al. show timings $10^4$ to $10^6$ times slower than that of computations on unencrypted data. Although it is unclear exactly which circuits are being computed, the timings given are in seconds, so we believe that HElib will not be a serious candidate for SCCC in the immediate future.

As reported in \cite{116}, the current performance of FHEW \cite{117} is poor compared with unencrypted operations. The authors report that FHEW processed a single homomorphic NAND operation followed by a re-encryption in 0.69s and using 2.2GB of RAM. Therefore, we also believe that FHEW is not a candidate for SCCC, as it currently stands.

Although claims regarding its performance have been made in the press \cite{315}, no benchmarking statistics have been made publicly available for Microsoft’s SEAL library \cite{198}. However, in \cite{5}, it is reported that, for SEAL v1, the time to perform one multiplication is approximately 140ms.
With regard to FV-NFLib \cite{99}, Bonte et al. \cite{57} recently reported a significant decrease in the time to evaluate a four layer Group Method of Data Handling (GMDH) neural network \cite{59} from 32s to 2.5s, as a result of their novel encoding of the inputs.

Aguilar-Melchor et al. \cite{5} report their experimental findings regarding HELib-MP \cite{283}. They show that HELib-MP outperforms FV-NFLib for large (2048-bit) plaintexts. They further go on to benchmark HELib-MP by computing RSA-2048 and ECC-ElGamal-P256. An exponentiation in RSA-2048 takes between 157ms and 1.8s depending on the window size and number of multiplications required. For ECC-ElGamal-P256, an elliptic curve multiplication takes between 96ms and 242ms depending on window size and number of elliptic curve additions.

Catalano et al. \cite{74} provide experimental results for their work. For 128-bit plaintexts, our algorithms are approximately 10 to 1000 times faster at performing a multiplication operation and our most complex algorithm, HE2N, is roughly equal to their fastest, an extension of Joye-Libert \cite{177}, for additions.

Yu et al. \cite{357} give experimental results for their implementation of the Zhou and Wornell scheme \cite{365}. From their Figures 3 to 5, it is hard to compare our scheme with theirs directly but it would appear that our vector based schemes are at least comparable in performance to theirs.

### 7.7.1 Microsoft Azure Cloud

In addition to performing our experiments in a small private cloud, we also scaled our experiment to a large cluster in Microsoft’s Azure public cloud \cite{228}. Our experimental environment consisted of a HDInsight cluster comprising two D13v2 head nodes and 123 D4v2 worker nodes (984 worker cores).

The number of inputs was increased to 106,272,000 for each experimental configuration. As a result of the large volume of input data, our experiments were altered to create and encrypt the data \textit{in situ}. This encrypted data is then consumed by our experimental program. As before, the output from our experiment was downloaded to a secure client and decrypted to verify the result. In addition, as a result of the larger number of inputs, we employed a tighter upper bound.

\footnote{This work was aided by a Microsoft Azure for Research sponsorship.}
166

CHAPTER 7. INTEGER HE FOR CRYPTMR

Table 7.2: Timings for each experimental configuration (n = 106272000 in all
cases, λ > 96). Init is the initialisation time for the encryption algorithm, Enc
is the mean time to encrypt a single integer, Exec is the total MR job execution
time, Prod is the mean time to homomorphically compute the product of two
encrypted integers, Sum is the mean time to homomorphically compute the sum
of two encrypted integers.
Alg.
d
HE1
HE1
HE1
HE1
HE1
HE1
HE1
HE1
HE1
HE1N
HE1N
HE1N
HE1N
HE1N
HE1N
HE1N
HE1N
HE1N
HE1N
HE1N
HE1N
HE1N
HE1N
HE1N
HE1N
HE2
HE2
HE2
HE2
HE2
HE2
HE2
HE2
HE2
HE2N
HE2N
HE2N
HE2N
HE2N
HE2N
HE2N
HE2N
HE2N
HE2N
HE2N
HE2N
HE2N
HE2N
HE2N
HE2N

2
2
2
3
3
3
4
4
4
2
2
2
2
2
2
2
2
3
3
3
3
3
3
3
3
2
2
2
3
3
3
4
4
4
2
2
2
2
2
2
2
2
3
3
3
3
3
3
3
3

Parameters
Encryption
MR Job
ρ
ρ0 Init(ms) Enc(µs) Exec(s) Prod(µs) Sum(µs)
32
64
128
32
64
128
32
64
128
1
1
1
8
8
8
16
16
1
1
1
8
8
8
16
16
32
64
128
32
64
128
32
64
128
1
1
1
8
8
8
16
16
1
1
1
8
8
8
16
16

85.53
94.9
108.41
103.05
133.23
146.93
110.88
157.18
2244.7
32
120.36
64
254.62
128
543.44
32
220.88
64
197.35
128
558.05
64
159.33
128
433.71
32
617.06
64 1656.53
128 43002.78
32 16850.53
64 2807.49
128 3701.99
64 18828.28
128 4672.69
122.25
116.27
131.78
152.55
180.11
214.22
160.11
257.11
- 1102.95
32
154.46
64
360.2
128
869.88
32
384.53
64
196.91
128
363.02
64
411.43
128
228.27
32
331.43
64 1197.62
128 2805.34
32 2658.76
64 1047.95
128 7375.87
64
850.94
128 7057.17

32.28
39.92
54.96
47.72
65.26
105.08
62.83
101.26
201.59
81.56
113.8
214.76
121.43
96.82
190.3
112.72
171.56
130.88
265
709.31
402.77
197.57
585.24
415.87
474.73
525.64
549.24
624.37
560.01
650.07
1462.87
658.37
1446.8
1921.06
677.64
1464.05
1992.68
1472.49
713.34
1803.51
764.95
1660.48
1553.09
5841.29
12600
5828.94
1852.06
7292.07
6375.15
6986.84

28.29
27.81
32.37
28.39
35.3
66.31
34.23
70.7
86.06
33.04
60.43
68.83
62.32
35.99
71.05
35.5
64.5
68.5
284
626.21
307.76
75.14
340.2
309.18
315.1
103.38
118.68
120.37
103.56
122.14
452.92
130.34
448.06
601.83
121.81
422.87
511.34
433.05
140.85
503.41
146.27
466.07
446.9
3120.64
7193.61
3317.31
537.16
4344.95
3359.67
3358.11

11.47
11.45
29.65
17.19
41.01
283.02
41.86
298.73
398.64
31.19
226.07
289.12
234.7
40.38
287.43
42.87
261.45
300.76
1520.65
3671.64
1659.24
352.67
1834.74
1696.96
1708.43
75.11
84.28
168.72
111.55
211.68
2423.63
219.3
2432.22
3147.91
197.3
2256.1
2896.89
2275.24
290.54
2789.1
336.36
2528.7
2450.82
16400
40300
17900
2996.51
19500
18300
17900

2.73
2.51
2.42
2.55
2.62
1.76
2.54
1.96
2.77
2.46
1.71
2.25
2.01
2.2
2.25
2.59
2.23
1.91
3.08
4.89
3.25
2.31
4.59
3.36
3.41
1.86
1.91
2.12
2.14
2.24
2.81
2.43
2.87
3.84
2.32
2.54
3.43
2.71
2.33
3.54
2.5
2.87
2.86
4.99
8.25
5.36
3.59
7.31
5.27
5.45

Decrypt
(ms)
245
647
315
380
347
555
561
601
27720
4757
6093
34792
5854
5371
35071
5697
33943
32622
41454
34996
33540
33591
37663
35064
38737
784
1308
1482
960
1650
2399
1687
2610
4200
1149
2889
3257
2515
1154
2740
1833
2565
5360
8129
12565
8676
3753
11014
10942
6026


(nM^d/d) on the values of \( p \) and \( \kappa \) required to successfully decrypt the HE1/HE2 and HE1N/HE2N variants respectively.

The results for each experimental configuration are presented in Table 7.2. Comparing it with Table 7.1 shows only an increase of approximately 300% in average encryption and product calculation times for our most complex algorithm (HE2N), despite the 4,428-fold increase in the number of inputs. The fact that these times did not scale linearly with the number of inputs may largely be attributed to the increased memory per worker node (28GB versus 2GB) and the tighter bounds on \( p \) and \( \kappa \). However, we also note that the bit size of the arbitrary precision integers involved will scale logarithmically in the number of inputs as a result of the tighter bound. Hence, the timings for arithmetic operations on those integers will also scale logarithmically.

### 7.8 Conclusion

In this chapter we have presented several new homomorphic encryption schemes intended for use in our CryptMR system. These schemes implement the HOM scheme we require for CryptMR to varying degrees of security and complexity. We envisage that the majority of computation on integer big data, outside of scientific computing, will be computing low degree polynomials on integers, or fixed-point decimals which can be converted to integers. Our somewhat homomorphic schemes are perfectly suited to these types of computation. We should note that our schemes only support homomorphic addition and multiplication, allowing for computation of polynomials, but do not support other operations homomorphically. This limits the scope of computations that can be performed using these encryption schemes. We should also note that these schemes are mutually inoperable with the schemes presented in chapters 8 and 9, so we cannot, say, meaningfully sort the output of a computation performed on data encrypted using the schemes presented in this chapter. We also note that our HEkN schemes can not be broken by Shor’s algorithm [302]. Should Shor’s algorithm can be used to factor the public modulus to obtain the secret key, and then the residues modulo \( p \) of the ciphertexts obtained, these residues still form an instance of GACDP as a result of the noise term. Therefore, HEkN has resistance to quantum cryptanalysis. For the HEk schemes, HE1 is immediately broken by the ability to
factorise the public modulus. The vector based schemes, HE2 and HE\(k\), still require an attacker to solve the system of equations for the plaintext values and the \(\mathbf{a}\) unknowns. However, with enough known plaintexts, an attacker is able to eliminate the plaintext terms and then solve for the \(\mathbf{a}\) terms. So, the HE\(k\) schemes are not resistant to quantum cryptanalysis. Finally, we observe that, by requirement to support homomorphic arithmetic operations, ciphertexts are mutable. Multiplying a ciphertext by a constant or adding a suitably large constant several times will result in the plaintext component of the ciphertext overflowing \(p\), the secret modulus. This would render the ciphertext undecryptable and could be used to disrupt a computation. However, in the context of MapReduce, ciphertext tampering can be detected by a result verification method such as our own method presented in chapter \([10]\). Similar tamper detection methods might be employed in other application contexts.

Our evaluation has only concerned one- or two-dimensional ciphertexts and polynomials of degree up to four. We intend to investigate higher degree polynomials in future work. We believe that HE1N and HE2N provide strong security, even for low-entropy data, as they satisfy the desirable IND-CPA property. If a user has a high confidence in the entropy of the input data and a static computation is performed, HE2 may provide sufficient security.

As they are only somewhat homomorphic, each of these schemes require that the computational result cannot grow bigger than the secret modulus. In the case of the “noise” variants, we also have to consider the noise term growing large. So, as they stand, these schemes can only compute polynomials of suitably bounded degree. However, we believe this is adequate for most practical purposes.

A further concern is that the ciphertext space is much larger than the plaintext space. This is as a result of adding multiples of large primes to the plaintext along with allowing for the multiplications of ciphertexts while still being able to decrypt the result. However, we note that this growth in the size of the ciphertext space is linear in the bit length of the plaintext space. Furthermore, we have shown that values exist which makes the system practical for computing low degree polynomials. Similar schemes \([97, 111]\) produce ciphertexts infeasibly larger than the corresponding plaintext, which is a single bit. For example, it should be noted, that even the practical CryptDB \([272]\), which is only additively homomorphic, enciphers a 32-bit integer as a 2048-bit ciphertext. Our schemes
will produce ciphertext of similar size, if high security is required. However, if the security is only intended to prevent casual snooping, rather than a determined cryptographic attack, the ciphertext size can be reduced. Observe that, for a static computation, as is the case in our usage scenario, the parameters of the system will change for each computation, so a sustained attack has constantly to re-learn these parameters. Of course, if the attacker is able to export data for off-line cryptanalysis, only high security suffices.

Our usage scenario only permits a static computation. If this is relaxed to allow for interactive computations or for computations on persistent data sources, then we must only use the HEkN variants because the HEk variants are all vulnerable to CPA and, hence, insecure (see the “Security” subsections of sections 7.2.1, 7.3.1, 7.4.1.

From the points made above, we see that there is trade-off between efficiency, security and space for our schemes. While our HEk are more efficient than the HEkN variants, particularly HE1, they are only secure in the context of static computations. The HEkN variants are IND-CPA secure but require larger space requirements and are less efficient as a result of the added noise term. If security is paramount, only the HEkN variants will suffice.

In the MapReduce context, we should note that our scheme have limitations regarding data aggregation in the shuffle and reduce phases. Our schemes produce ciphertexts that do not preserve any ordering or equality on the plaintexts. Therefore, if the key field of a MapReduce key-value pair were encrypted by any of our schemes it would not be aggregated by the shuffle phase in any meaningful way. Also, as our encryption schemes only support homomorphic additions and multiplications, the reduce function cannot perform other operations on the encrypted data.

The schemes presented in sections 7.2 and 7.3 extend to a hierarchy of systems, HEk, with increasing levels of security. These are presented in section 7.4 and may be investigated further in future work. As stated in section 7.5, we can extend HE2NCRT to k-vectors to create a HEkNCRT scheme. The discussion will appear in future work. We also showed that our HEk system can be extended to an FHE system. We may also investigate several enhancements of this FHE system. First, we could implement a packed ciphertext optimisation for our scheme where an operation is performed on a vector of ciphertexts rather than
a single ciphertext. We can also investigate an improvement to address circuit privacy. Finally, we can investigate applying the Chinese Remainder Theorem secret sharing method employed in HE2NCRT (section 7.5) to our fully homomorphic scheme (section 7.6).

We have implemented and evaluated the HE1, HE1N, HE2 and HE2N schemes as part of an SSCC system as discussed in section 7.7. Our results are extremely favourable by comparison with existing methods. In some cases, they outperform those methods by a factor of 1000. We have also performed extensive experiments with large data sets (approximately 100 million inputs) and shown that the increase in time to perform encryption and arithmetic operations grows sublinearly in the number of inputs. This clearly demonstrates the practical applicability of our schemes. Furthermore, our MapReduce job execution times remain low even when using the largest set of parameters for HE2N. We believe that this demonstrates the advantages of our schemes for encrypted computations on fixed-point data in the cloud.

To conclude, we believe that the work in this chapter, has achieved our goal of providing a HOM scheme for CryptMR. In particular, our work significantly outperforms related work with regard to the running time of arithmetic operations on encrypted data.
Chapter 8

Order-Preserving Encryption for CryptMR

The material in this chapter, excepting section 8.4.1, was presented at the 12th International Workshop on Data Privacy Management (DPM 2017) [120].

8.1 Introduction

For sorting and comparison of data CryptMR requires an encryption scheme that supports homomorphic comparisons of ciphertexts. Such a scheme would not only allow comparisons on encrypted data in the user-defined map and reduce functions but would also allow the MapReduce framework to correctly sort encrypted data during the shuffle phase. As we discussed in 3.2.3, order-preserving encryption (OPE) is a recent field that supports just such a proposition. An OPE is defined as an encryption scheme where, for plaintexts \( m_1 \) and \( m_2 \) and corresponding ciphertexts \( c_1 \) and \( c_2 \):

\[
 m_1 < m_2 \implies c_1 < c_2
\]

Our work here presents an OPE scheme that is based on the general approximate common divisor problem (GACDP) [164], which is believed to be hard. Using this

\footnote{This relationship is typically represented as \( m_1 \leq m_2 \implies c_1 \leq c_2 \). However, this seems to introduce an insecurity, by permitting an equality test for plaintexts using two comparisons.}
problem we have devised a system where encryption and decryption require $O(1)$
arithmetic operations. In section 8.2, we present our OPE scheme. In section
8.3 we provide the generic version of Boldyreva et al.’s algorithm and the Beta
distribution approximation used in our experiments. In section 8.4 we discuss
the results of experiments on our OPE scheme. Finally, in section 8.5 we conclude
this chapter.

8.1.1 Formal Model of Scenario

The model of section 6.2.2 applies with the following specification. Our $n$ inputs,
$m_1, m_2, \ldots, m_n$, are integers where $m_i \in \mathcal{M} = [0, M]$ and $n \ll M$. The function
$f$ we wish to compute on the inputs is their ordering. The cloud environment
conducts comparisons on the $c_i, i \in [1, n]$. Since Enc is an OPE, the $m_i$ will
also be correctly sorted. $A$ can retrieve some or all of the $c_i$ from the cloud and
decrypt each ciphertext $c_i$ by computing $m_i = \text{Dec}(K, c_i)$.

8.1.2 Further Observations from Scenario

In addition to the observations from 6.2.3, we also note that with regard to CPA,
it can be argued that any notion of indistinguishability under CPA is not rel-
vant to OPE in practice (see section 8.2.2). Various attempts have been made
by Boldyreva and others [52, 53, 312, 354] to provide such indistinguishability
notions. However, the security models impose practically unrealistic restrictions
on an adversary. See, for example, our discussion of IND-OCPA below (section
8.2.2). It should also be pointed out that satisfying an indistinguishability cri-
terion does not guarantee that a cryptosystem is unbreakable, and neither does
failure to satisfy it guarantee that the system is breakable as we discuss in chapter
5.

8.1.3 Related Works

Unlike the schemes of [52, 53, 196, 312], which are deterministic, our OPE scheme
is randomised. This has the advantage that multiple encryptions of a plaintext

\footnote{We must assume $n \ll M$ to avoid the “sorting attack” of Naveed et al. [212]}
8.2. AN OPE SCHEME USING APPROXIMATE COMMON DIVISORS

will produce differing ciphertexts.

Liu and Wang [211] describe a system similar to ours where random “noise” is added to a linear transformation of the plaintext. The security of their scheme is unclear, particularly because they use real numbers. The security of our system is derived from a computationally hard problem.

Unlike Khadem et al. [179], our scheme allows for non-consecutive integers. This means that our scheme can support updates without worrying about overlapping “buckets” as Khadem et al.

Like Agrawal [4] and Yum et al. [358], we discuss a technique to “flatten” the data in section 8.2.3. This mitigates against the situation where an attacker has knowledge of the distribution of the plaintexts.

As discussed in section 3.2.3, the number of inputs, $n$, must be much smaller than the size of plaintext space, $M$. In our implementations of Boldyreva et al.’s algorithm [53], we use a ciphertext space size, $N \geq M^2$, since this has the advantage that the scheme can be approximated closely by a much simplified computation, as we discuss in section 8.3.2. The cost is only a doubling of the ciphertext size.

ORE schemes (section 3.2.3) have been proposed as an alternative to OPE. Lewi and Wu [202] give experimental results for their ORE scheme. Our results (8.4) compare favourably with theirs. This is largely because the ciphertext sizes of Lewi and Wu’s scheme are much larger. Furthermore, as discussed in 3.2.3, ORE schemes such as Lewi and Wu permit an equality test. A randomised OPE scheme, like ours, does not permit this.

8.2 An OPE scheme using Approximate Common Divisors

Our OPE scheme is the symmetric encryption system ($\text{KGen}$, $\text{Enc}$, $\text{Dec}$). The message space, $\mathcal{M}$, is $[0, M]$, and the ciphertext space, $\mathcal{C}$, is $[0, N]$, where $N > M$. We have plaintexts $m_i \in \mathcal{M}, i \in [1, n]$ such that $0 < m_1 \leq m_2 \leq \cdots \leq m_n \leq M$. 
Key Generation.

Both the security parameter space $S$ and the secret key space $K$ are the set of positive integers. Given a security parameter $\lambda \in S$, with $\lambda > 8/3 \lg M$, Algorithm 8.1 randomly chooses an integer $k \in [2^\lambda, 2^{\lambda+1})$ as the secret key, $sk$. So $k$ is a $(\lambda + 1)$-bit integer such that $k > M^{8/3}$ (see section 8.2.1). Note that $k$ does not necessarily need to be prime.

```
Algorithm 8.1: Key Generation Algorithm KGen

| Input : $\lambda \in S$, $\lambda > 8/3 \lg M$ |
| Output: $k \in K$ |
| 1 $k \leftarrow [2^\lambda, 2^{\lambda+1})$; |
| 2 return $k$; |
```

Encryption.

A plaintext $m_i \in M$ is encrypted by Algorithm 8.2.

```
Algorithm 8.2: Encryption Algorithm Enc

| Input : $m_i \in M$ |
| Input : $k \in K$ |
| Output: $c_i \in C$ |
| 1 $r_i \leftarrow (k^{3/4}, k - k^{3/4})$; |
| 2 $c_i \leftarrow m_i k + r_i$; |
| 3 return $c_i$; |
```

Decryption.

A ciphertext $c_i \in C$ is decrypted by Algorithm 8.3.

Order-preserving property.

If $m > m'$, then $c \geq c'$ provided $mk + r > m'k + r'$, i.e. if $k(m - m') > (r' - r)$, which follows, since the lhs is at least $k$, and the rhs is less than $(k-1)$. If $m' = m$, then the order of the encryptions is random, since $\Pr(r' > r) \approx \frac{1}{2} - 1/k \approx \frac{1}{2}$. 
Algorithm 8.3: Decryption Algorithm Dec

<table>
<thead>
<tr>
<th>Input</th>
<th>$c_i \in C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>$k \in K$</td>
</tr>
<tr>
<td>Output</td>
<td>$m_i \in M$</td>
</tr>
</tbody>
</table>

1. $m_i \leftarrow \lfloor c_i/k \rfloor$;
2. return $m_i$;

8.2.1 Security of the Scheme

Security of our scheme is given by the general approximate common divisor problem (GACDP), which is believed to be hard. It can be formulated \(^7\)\(^8\)\(^9\)\(^2\) as:

**Definition 24 (General approximate common divisor problem).** Suppose we have $n$ integer inputs $c_i$ of the form $c_i = km_i + r_i$, $i \in [1, n]$, where $k$ is an unknown constant integer and $m_i$ and $r_i$ are unknown integers. We have a bound $B$ such that $|r_i| < B$ for all $i$. Under what conditions on $m_i$ and $r_i$, and the bound $B$, can an algorithm be found that can uniquely determine $k$ in a time which is polynomial in the total bit length of the numbers involved?

GACDP and partial approximate common divisor problem (PACDP), its close relative, are used as the basis of several cryptosystems, e.g. \(^97\)\(^111\)\(^121\). Solving the GACDP is clearly equivalent to breaking our system. To make the GACDP instances hard, we need $k \gg M$ (see below). Furthermore, we need the $m_i$ to have sufficient entropy to negate a simple “guessing” attack \(^216\) as we previously discussed in section \(^3.2.3\). However, note that the model in \(^216\) assumes that we are able to verify when a guess is correct, which does not seem to be the case here. Although our scenario does not permit it, even if we knew a plaintext, ciphertext pair $(m, c)$, it would not allow us to break the system, since $c/m = k+r/m \in [k, k+k/m]$, which is a large interval since $k \gg M$. We note that small values of $m$ reveal much less information than large values. A number $n$ of such pairs would give more information, but it still does not seem straightforward to estimate $k$ closer than $\Omega(k/(M\sqrt{n}))$. Thus the system has some resistance to KPA, even though this form of attack is excluded by our model of single-party secure computation.

Howgrave-Graham \(^164\) studied two attacks against GACD, to find divisors $d$ of $a_0 + x_0$ and $b_0 + y_0$, given inputs $a_0, b_0$ of similar size, with $a_0 < b_0$. The
quantities \( x_0, y_0 \) are the “offsets”. The better attack in \([164]\), \text{GACD}\_L, succeeds when \( |x_0|, |y_0| < X = b_0^{\beta_0} \), and the divisor \( d \geq b_0^{\alpha_0} \) and

\[
\beta_0 = 1 - \frac{1}{2} \alpha_0 - \sqrt{1 - \alpha_0 - \frac{1}{2} \alpha_0^2 - \epsilon}.
\]

where \( \epsilon > 0 \) is a (small) constant, such that \( 1/\epsilon \) governs the number of possible divisors which may be output. We will take \( \epsilon = 0 \). This is the worst case for Howgrave-Graham’s algorithm, since there is no bound on the number of divisors which might be output.

Note that \( \beta_0 < \alpha_0 \), since otherwise \( \sqrt{1 - \alpha_0 - \frac{1}{2} \alpha_0^2} \leq 1 - \frac{3}{2} \alpha_0 \). This can only be satisfied if \( \alpha_0 \leq \frac{2}{3} \). But then squaring both sizes of the inequality implies \( \alpha_0 \geq \frac{8}{11} > \frac{2}{3} \), contradicting \( \alpha_0 \leq \frac{2}{3} \).

Suppose we take \( \alpha_0 = \frac{8}{11} \). Then, to foil this attack, we require \( \beta_0 \geq \frac{6}{11} \). For our system we have, \( b_0 - a_0 = \max m_i - \min m_i = M \) \[3\]. To ensure that the common divisor \( k \) will not be found we require \( b_0^{\alpha_0} \geq k \), so we will take \( k = b_0^{8/11} \). Since \( b_0 \sim Mk \), this then implies \( b_0 = M^{11/3} \). Thus the ciphertexts will then have about \( 11/3 \) times as many bits as the plaintexts. Now \text{GACD}\_L could only succeed for offsets less than \( b_0^{\beta_0} = b_0^{6/11} = k^{3/4} \). Thus, we choose our random offsets in the range \((k^{3/4}, k - k^{3/4})\).

Cohn and Heninger \[92\] give an extension of Howgrave-Graham’s algorithm to find the approximate divisor of \( m \) integers, where \( m > 2 \). Unfortunately, their algorithm is exponential in \( m \) in the worst case, though they say that it behaves better in practice. On the other hand, \[77\], Appendix A] claims that Cohn and Heninger’s algorithm is worse than brute force in some cases. In our case, the calculations in \[92\] do not seem to imply better bounds than those derived above.

We note also that the attack of \[78\] is not relevant to our system, since it requires smaller offsets, of size \( O(\sqrt{k}) \), than those we use.

For a survey and evaluation of the above and other attacks on GACD, see \[133\].

\[3\] Note this is our \( M \), not Howgrave-Graham’s.
8.2.2 Security Models

One-Wayness.

As we discussed in chapter 3, a one-way function is a function which is easy to compute but it is hard to compute the inverse function on a random input. The one-wayness of the function \( c(m) = km + r \) used by the scheme clearly follows from the assumed hardness of the GACD problem, since we avoid the known polynomial-time solvable cases.

IND-OCPA.

The model in [53, p.6] and [202, p.20] is as follows:

**Definition 25** (Indistinguishability under ordered CPA (IND-OCPA)). Given two equal-length sequences of plaintexts \((m_0^1 \ldots m_0^q)\) and \((m_1^1 \ldots m_1^q)\), where the \(m_j^b\) (\(b \in [0,1], j \in [1,q]\)) are distinct, an adversary is allowed to present two plaintexts to a left-or-right oracle \([28], \text{LR}^{(m_0,m_1,b)}\), which returns the encryption of \(m_b\). The adversary is only allowed to make queries to the oracle which satisfy \(m_i^0 < m_i^1\) iff \(m_j^0 < m_j^1\) for \(1 \leq i, j \leq q\). The adversary wins if it can distinguish the left and right orderings with probability significantly better than \(1/2\).

However, Boldyreva et al. [53, p.5] note, concerning chosen plaintext attacks: “in the symmetric-key setting a real-life adversary cannot simply encrypt messages itself, so such an attack is unlikely to be feasible”. Further, they prove that no OPE scheme with a polynomial size message space can satisfy IND-OCPA. Lewi et al. [202] strengthen this result under certain assumptions.

The IND-OCPA model seems inherently rather impractical, since an adversary with an encryption oracle could decrypt any ciphertext by bisection using \(\lg M\) comparisons, where \(M\) is the size of the message space. Furthermore, Xiao and Yen [353] construct an OPE for the domain \([1,2]\) and prove that it is IND-OCPA secure. However, this system is trivially breakable using a “sorting” attack [242]. For these reasons, we do not consider security models assuming CPA to be relevant to OPE.

---

4 [53, p.6] and [202, p.20] do not clearly state this assumption but it appears that all plaintext values used must be distinct. This assumption clearly does not weaken the model.
Window One-Wayness.

We may further analyse our scheme under the same model as in [52], which was called window one-wayness. The scenario is as follows.

Definition 26 (Window one-wayness). An adversary is given the encryptions $c_1 \leq c_2 \leq \cdots \leq c_n$ of a sample of $n$ plaintexts $m_1 \leq m_2 \leq \cdots \leq m_n$, chosen uniformly and independently at random from the plaintext space $[0, M)$. The adversary is also given the encryption $c$ of a challenge plaintext $m$, and must return an estimate $\hat{m}$ of $m$ and a bound $r$, such that $m \in (\hat{m} - r, \hat{m} + r)$ with probability greater than $1/2$, say. How small can $r$ be so that the adversary can meet the challenge?

This model seems eminently reasonable, except for the assumption that the plaintexts are distributed uniformly. However, as we show in section 8.2.3, this assumption can be weakened in some cases for our scheme.

Since the $m_i$ are chosen uniformly at random, a random ciphertext satisfies, for $c \in [0, kM)$,

$$\Pr(c = c) = \Pr(km + r = km + r) = \Pr(m = m) \Pr(r = r) = \frac{1}{M} \frac{1}{k} = \frac{1}{Mk},$$

where $m \xleftarrow{\$} [0, M)$, $r \xleftarrow{\$} [0, k)$. Thus $c$ is uniform on $[0, kM)$. Note that this is only approximately true, since we choose $r$ uniformly from $[k^{3/4}, k-k^{3/4}]$. However, the total variation distance between these distributions is $2Mk^{3/4}/Mk = 2/k^{1/4}$. The difference between probabilities calculated using the two distributions is negligible, so we will assume the uniform distribution.

By assumption, the adversary cannot determine $k$ by any polynomial time computation. So the adversary can only estimate $k$ from the sample. Now, in a uniformly chosen sample $c_1 \leq c_2 \leq \cdots \leq c_n$ from $[0, kM)$, the sample maximum $c_n$ is a sufficient statistic for the range $kM$, so all information about $k$ is captured by $c_n$. So we may estimate $k$ by $\hat{k} = c_n/M$. This is the maximum likelihood estimate, and is consistent but not unbiased. The minimum variance unbiased estimate is $(n + 1)\hat{k}/n$, but using this does not improve the analysis, since the bias $k/(n+1)$ is of the same order as the estimation error, as we now prove. For
8.2. AN OPE SCHEME USING APPROXIMATE COMMON DIVISORS

any \(0 \leq \varepsilon \leq 1\),

\[
\Pr \left( \hat{k} \in k(1 \pm \varepsilon) \right) = \Pr \left( c_n \geq kM(1 - \varepsilon) \right) \\
= 1 - (1 - \varepsilon)^n \quad \begin{cases} 
\leq n\varepsilon < \frac{1}{2} & \text{if } \varepsilon < 1/(2n); \\
\geq 1 - e^{-n\varepsilon} \geq \frac{1}{2} & \text{if } \varepsilon \geq \ln 2/n.
\end{cases}
\]

Now, if \(c = mk + r\), we can estimate \(m\) by \(\hat{m} = c/\hat{k} \approx mk/\hat{k}\). Then

\[
\Pr \left( m \in \hat{m}(1 \pm \varepsilon) \right) \approx \Pr \left( m \in mk/\hat{k}(1 \pm \varepsilon) \right) = \Pr \left( \hat{k} \in k(1 \pm \varepsilon) \right) < \frac{1}{2},
\]

if \(\varepsilon < 1/(2n)\). Thus, if \(r \leq m/2n\), \(\Pr(m \in \hat{m} \pm r) < \frac{1}{2}\). Similarly, if \(r \geq m \log 2/n\), \(\Pr(m \in \hat{m} \pm r) \geq \frac{1}{2}\). Thus the adversary cannot succeed if \(r \leq m/2n\), but can if \(r \geq m \log 2/n\).

It follows that only \(\log m - \log(m/n) + O(1) = \log n + O(1)\) bits of \(m\) are leaked by the system. However, \(\log n\) bits are leaked by inserting \(c\) into the sequence \(c_1 \leq c_2 \leq \cdots \leq c_n\), so the leakage is close to minimal. By contrast the scheme of [53] leaks \(\frac{1}{2} \log m + O(1)\) bits, independently of \(n\). Therefore, by this criterion, the scheme given here is superior to that of [53] for all \(n \ll \sqrt{M}\). Note that we have not assumed that \(m\) is chosen uniformly from \([0, M]\), but the leakage of the random sequence \(c_1 \leq c_2 \leq \cdots \leq c_n\) is clearly \(n \log n - O(n)\) of the \(M \log M\) plaintext bits. This reveals little more than the \(n \log n\) bits revealed by the known order \(m_1 \leq m_2 \leq \cdots \leq m_n\).

8.2.3 Further Observations

This scheme can be used in conjunction with any other OPE method, i.e. any unknown increasing function \(f(m)\) of \(m\). We might consider any integer-valued increasing function, e.g. a polynomial function of \(m\), or Boldyreva et al.’s scheme. If \(f(m)\) is this function, then we encrypt \(m\) by \(c = f(m)k + r\), where \(r \leftarrow \mathcal{S}(k^{3/4}, k - k^{3/4})\), and decrypt by \(m = f^{-1}([c/k])\). The disadvantage is that the ciphertext size will increase.

If \(f(m)\) is an unknown polynomial function, we solve a polynomial equation to decrypt. The advantage over straight GACD is that, even if we can break the GACD instance, we still have to solve an unknown polynomial equation to break
the system. For example, suppose we use the linear polynomial $f(m) = a_1(m + a_0) + s$, where $s \leftrightarrow [0, a_0]$ is random noise. But this gives $c = a_1k(m + a_0) + (ks + r)$, where $s \leftarrow [0, a_0]$ is random noise. But this gives $c = a_1k(m + a_0) + (ks + r)$, which is our OPE system with a deterministic linear monic polynomial $f(m) \leftarrow m + a_0$, $k \leftarrow a_1k$ and $r \leftarrow ks + r \leftrightarrow [0, a_1k]$, so $f(m)$ contains a single unknown parameter, $a_0$. More generally, we need only consider monic polynomials, for the same reason.

If $c = f(m)$ is Boldyreva et al.’s OPE, we can invert $f$ only with error $O(\sqrt{m})$. Therefore a hybrid scheme offers greater security than either alone.

**Flattening.**

Another use of such a transformation is when the distribution function $F(m)$ of the plaintexts is known, or can be reasonably estimated. Then the distribution of the plaintexts can be “flattened” to an approximate uniform distribution on a larger set $[0, N)$, where $N \gg M$. Thus, suppose the distribution function $F(m)$ $(M \in [0, M))$ is known, and can be computed efficiently for given $m$. Further, we assume that $\Pr(m = m) \geq 1/N$, so $F$ is strictly increasing. This assumption is weak, since the probability that $m$ is chosen to be an $m$ with too small probability is at most $M/N$, which we assume to be negligible.

We interpolate the distribution function linearly on the real interval $\mathbb{R}[0, M)$, by $F(x) = (1 - u)F(m) + uF(m + 1)$ for $x = (1 - u)m + u(m + 1)$, where $u \in \mathbb{R}[0, 1)$. Then we will transform $m \in [0, M)$ randomly by taking $\tilde{m} = NF(x)$ where $u$ is chosen randomly from the continuous uniform distribution on $\mathbb{R}[0, 1)$. It follows that $\tilde{m}$ is uniform on $\mathbb{R}[0, N)$, since $F$ is increasing, and $\tilde{m} = NF(x)$, since

$$\Pr(\tilde{m} \leq y) = \Pr(x \leq F^{-1}(y/N)) = F(F^{-1}(y/N)) = y/N.$$ 

Now, since we require a discrete distribution, we take $\tilde{m} = \lfloor \tilde{m} \rfloor$. We invert this by taking $\hat{m} = \lfloor F^{-1}(\tilde{m}) \rfloor$. Now, since $F$ is strictly increasing, $\hat{m} = \lfloor F^{-1}(\tilde{m}/N) \rfloor \leq F^{-1}(\tilde{m}/N) < F^{-1}(NF(m + 1)/N) = m + 1$ and so $\hat{m} = m$. Thus the transformation is uniquely invertible. Of course, this does not imply that $\hat{m}$ and $m$ will have exactly the same distribution, but we
may also calculate
\[
\Pr(\hat{m} \leq x) \leq \Pr(\hat{m} \leq NF(x)) < Pr(\hat{m} \leq NF(x) + 1) = F(x) + 1/N, \\
Pr(\hat{m} \leq x) \geq \Pr(\hat{m} < NF(x + 1)) \geq Pr(\hat{m} < NF(x)) = F(x).
\]

This holds, in particular, for integers \(x \in [0, M]\). Thus the total variation distance between the distributions of \(\hat{m}\) and \(\hat{m}\) is at most \(M/N\). Thus the difference between the distributions of \(\hat{m}\) and \(\hat{m}\) will be negligible, since \(N \gg M\).

This flattening allows us to satisfy the assumptions of the window one-wayness scenario above. The bit leakage in \(m\) is increased, however.

**Theorem 15.** The increase in bit leakage for \(m\) as a result of this flattening is approximately \(\log(mp_m/F(m))\), where \(p_m = F(m) - F(m-1)\).

The leakage remains near-optimal for near-uniform distributions, where \(\alpha/M \leq p_m \leq \beta/M\), for some constants \(\alpha, \beta > 0\). In this case \(\log(mp_m/F(m)) \leq \log(\beta/\alpha) = O(1)\). There are also distributions which are far from uniform, but the ratio \(mp_m/F(m)\) remains bounded. Further, suppose we have a distribution satisfying \(1/m^\alpha \leq p_m \leq 1/m^\beta\), for constants \(\alpha, \beta > 0\) such that \(0 < \alpha - \beta < 1/2\). Then \(\log(mp_m/F(m)) < 1/2 \log m\), so the leakage is less than in the scheme of [53].

This transformation also allows us to handle relatively small plaintext spaces \([0, M]\), by expanding them to a larger space \([0, N]\).

Finally, note that the flattening approach here is rather different from those in [4] and [358], though not completely unrelated.

### 8.3 Algorithms of Boldyreva type

We have chosen to compare our scheme with that of Boldyreva et al. [53], since it has been used in practical contexts by the academic community [52, p.5], as well as in Popa et al.’s original version of CryptDB [272], which has been used or adopted by several commercial organisations [273]. However, scant computational experience with the scheme has been reported [272]. Therefore, we believe it is of academic interest to report our experimental results with respect to Boldyreva
et al.’s scheme. We also discuss some simpler variants which have better computational performance. These are compared computationally with our scheme in section 8.4 below. The relative security of the schemes has been discussed above.

In this section we describe generic encryption and decryption algorithms based on Boldyreva et al.’s algorithm \[53\], which sample from any distribution and which bisect on the domain (section 8.3.1). We also present an approximation of Boldyreva et al.’s algorithm which samples from the Beta distribution (section 8.3.2). The approximation and generic algorithms are used in our experimental evaluation presented in section 8.4.

### 8.3.1 Generic Algorithms

Algorithm 8.4 below constructs a random order-preserving function $f : \mathcal{M} \rightarrow \mathcal{C}$, where $\mathcal{M} = [0, M], M = 2^r$, and $\mathcal{C} = [1, N], N \geq 2^{2r}$, so that $c = f(m)$ is the ciphertext for $m \in \mathcal{M}$. Algorithm 8.4 depends on a pseudorandom number generator, $P$, and a deterministic seed function, $S$. Likewise, Algorithm 8.5 constructs the inverse function $f^{-1} : \mathcal{C} \rightarrow \mathcal{M}$ so that $m = f^{-1}(c)$.

**Algorithm 8.4: Generic Boldyreva-type Encryption Algorithm**

<table>
<thead>
<tr>
<th>Line</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Function RecursiveEncrypt($a, b, f(a), f(b), m$)</td>
</tr>
<tr>
<td>2</td>
<td>$x \leftarrow (a + b)/2$</td>
</tr>
<tr>
<td>3</td>
<td>$y \leftarrow f(b) - f(a)$</td>
</tr>
<tr>
<td>4</td>
<td>Initiate $P$ with seed $S(a, b, f(a), f(b))$</td>
</tr>
<tr>
<td>5</td>
<td>Determine $z \in [0, y]$ pseudorandomly, so that $\Pr(z \notin [y/4, 3y/4])$ is negligible. // The condition implies that $y$ cannot become smaller than $3N/4(3/4)^r = 3N/4M^2 = 3M/4$, with high probability.</td>
</tr>
<tr>
<td>6</td>
<td>$f(x) \leftarrow f(a) + z$</td>
</tr>
<tr>
<td>7</td>
<td>if $x = m$ then return $f(x)$</td>
</tr>
<tr>
<td>8</td>
<td>else if $x &gt; m$ then return RecursiveEncrypt($a, x, f(a), f(x), m$)</td>
</tr>
<tr>
<td>9</td>
<td>else return RecursiveEncrypt($x, b, f(x), f(b), m$)</td>
</tr>
<tr>
<td>10</td>
<td>Initiate $P$ with a fixed seed $S_0$</td>
</tr>
<tr>
<td>11</td>
<td>Choose $f(0), f(M)$ pseudorandomly so that $f(M) - f(0) &gt; 3N/4$</td>
</tr>
<tr>
<td>12</td>
<td>return RecursiveEncrypt($0, M, f(0), f(M), m$)</td>
</tr>
</tbody>
</table>
8.3. **ALGORITHMS OF BOLDYREVA TYPE**

<table>
<thead>
<tr>
<th>Algorithm 8.5: Generic Boldyreva-type Decryption Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Function RecursiveDecrypt( (a, b, f(a), f(b), c) )</td>
</tr>
<tr>
<td>2 ( x \leftarrow (a + b)/2 )</td>
</tr>
<tr>
<td>3 ( y \leftarrow f(b) - f(a) )</td>
</tr>
<tr>
<td>4 Initiate ( P ) with seed ( S(a, b, f(a), f(b)) )</td>
</tr>
<tr>
<td>5 Determine ( z \in [0, y] ) pseudorandomly</td>
</tr>
<tr>
<td>6 ( f(x) \leftarrow f(a) + z )</td>
</tr>
<tr>
<td>7 if ( f(x) = c ) then return ( x )</td>
</tr>
<tr>
<td>8 else if ( f(x) &gt; c ) then return ( ) RecursiveDecrypt( (a, x, f(a), f(x), c) )</td>
</tr>
<tr>
<td>9 else ( ) return ( ) RecursiveDecrypt( (x, b, f(x), f(b), c) )</td>
</tr>
<tr>
<td>10 Initiate ( P ) with a fixed seed ( S_0 )</td>
</tr>
<tr>
<td>11 Choose ( f(0), f(M) ) pseudorandomly so that ( f(M) - f(0) &gt; 3N/4 )</td>
</tr>
<tr>
<td>12 return ( ) RecursiveDecrypt( (0, M, f(0), f(M), c) )</td>
</tr>
</tbody>
</table>

### 8.3.2 An Approximation

We have a plaintext space, \([1, M]\), and ciphertext space, \([1, N]\). Boldyreva et al. use bijection between strictly increasing functions \([1, M] \rightarrow [1, N]\) and subsets of size \(M\) from \([1, N]\), so there are \(\binom{N}{M}\) such functions. There is a similar bijection between nondecreasing functions \([1, M] \rightarrow [1, N]\) and multisets of size \(M\) from \([1, N]\), and there are \(N^M/M!\) such functions. If we sample \(n\) points from such a function \(f\) at random, the probability that \(f(m_1) = f(m_2)\) for any \(m_1 \neq m_2\) is at most \(\left(\frac{n}{2}\right) \times 1/N < n^2/2N\). We will assume that \(n < \sqrt{N}\), so \(n^2/2N\) is negligible. Hence we can use sampling either with or without replacement, whichever is more convenient.

Suppose we have sampled such a function \(f\) at points \(m_1 < m_2 < \cdots < m_k\), and we now wish to sample \(f\) at \(m\), where \(m_i < m < m_{i+1}\). We know \(f(m_i) = c_i\), \(f(m_{i+1}) = c_{i+1}\), and let \(f(m) = c\), so \(c_i \leq c \leq c_{i+1}\)\(^5\). Let \(x = m - m_i\), \(a = m_{i+1} - m_i - 1\), \(b = c_{i+1} - c_i - 1\), so \(1 \leq x \leq a\) and \(0 \leq y \leq b\). Write \(\hat{f}(x) = f(x + m_i) - c_i\). Then, if we sample \(a\) values from \([0, b]\) independently and uniformly at random, \(c - c_i\) will be the \(x\)th smallest. Hence we may calculate, for \(0 \leq y \leq b\),

\[
\Pr(\hat{f}(x) = y) = \frac{a!}{(x-1)! \cdot (a-x)!} \left(\frac{y}{b}\right)^{x-1} \left(\frac{b-y}{b}\right)^{a-x}
\]

\(^5\)We can have equality because we sample with replacement.
CHAPTER 8. OPE FOR CRYPTMR

This is the probability that we sample one value \( y \), \((x - 1)\) values in \([0, y]\) and \((a - x)\) values in \((y, b]\), in any order. If \( b \) is large, let \( z = y/b \), and \( dz = 1/b \), then (8.1) is approximated by a continuous distribution with, for \( 0 \leq z \leq 1 \),

\[
\Pr\left( z \leq \tilde{f}(x)/b < z + dz \right) = \frac{z^{x-1}(1 - z)^{a-x}}{B(x, a - x + 1)} \, dz
\]  

(8.2)

which is the \( \text{B}(x, a - x + 1) \) distribution. Thus we can determine \( f(m) \) by sampling from the Beta distribution to \( \lg N \) bits of precision. In fact, we only need \( \lg b \) bits. However, using \( n \leq M \leq \sqrt{N} \),

\[
\Pr(\exists i : m_{i+1} - m_i < N^{1/3}) \leq \frac{nN^{1/3}}{N} \leq \frac{M}{N^{2/3}} \leq \frac{1}{N^{1/6}}
\]

is very small, so we will almost always need at least \( \frac{1}{3} \lg N \) bits of precision. Thus the approximation given by (8.2) remains good even when \( a = 1 \), since it is then the uniform distribution on \([0, b]\), where \( b \geq N^{1/3} \) with high probability.

When the \( m_i \) arrive in random order, the problem is to encrypt them consistently without storing and sorting them. Boldyreva et al. use binary search. If \( M = 2^r \), we will always have \( a = 2^s \) and \( x = 2^{s-1} \) in (8.2), so \( a - x = x \), and (8.2) simplifies to

\[
\Pr\left( z \leq \tilde{f}(x)/b < z + dz \right) = \frac{z^{x-1}(1 - z)^x}{B(x, x + 1)} \, dz,
\]

for \( 0 \leq z \leq 1 \), This might be closely approximated by a Normal distribution if Beta sampling is too slow.

8.4 Experimental Results

To evaluate our scheme in practice, we conducted a simple experiment to pseudorandomly generate and encrypt 10,000 \( \rho \)-bit integers. The ciphertexts were then sorted using a customised TeraSort MapReduce (MR) algorithm [249]. Finally, the sorted ciphertexts were decrypted and it was verified that the plaintexts were also correctly sorted.

The MR algorithm was executed on a Hadoop cluster of one master node and 16 slaves. Each node was a Linux virtual machine (VM) having 1 vCPU and 2GB
Table 8.1: Timings for each experimental configuration \((n = 10000)\). \(\rho\) denotes the bit length of the unencrypted inputs. \(\text{Init}\) is the initialisation time for the encryption/decryption algorithm, \(\text{Enc}\) is the mean time to encrypt a single integer, \(\text{Exec}\) is the MR job execution time, \(\text{Dec}\) is the mean time to decrypt a single integer.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>(\rho)</th>
<th>Encryption Init. (ms)</th>
<th>Encryption Enc. (µs)</th>
<th>MR Job Exec. (s)</th>
<th>Decryption Init. (ms)</th>
<th>Decryption Dec. (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>GACD</td>
<td>7</td>
<td>50.13</td>
<td>1.51</td>
<td>63.79</td>
<td>11.62</td>
<td>1.47</td>
</tr>
<tr>
<td>GACD</td>
<td>15</td>
<td>58.04</td>
<td>2.18</td>
<td>61.28</td>
<td>10.86</td>
<td>2.46</td>
</tr>
<tr>
<td>GACD</td>
<td>31</td>
<td>58.66</td>
<td>2.07</td>
<td>63.02</td>
<td>12.18</td>
<td>2.59</td>
</tr>
<tr>
<td>GACD</td>
<td>63</td>
<td>70.85</td>
<td>1.94</td>
<td>65.20</td>
<td>10.61</td>
<td>4.22</td>
</tr>
<tr>
<td>GACD</td>
<td>127</td>
<td>91.94</td>
<td>2.38</td>
<td>61.08</td>
<td>11.10</td>
<td>6.29</td>
</tr>
<tr>
<td>BCLO</td>
<td>7</td>
<td>143.72</td>
<td>191.48</td>
<td>70.78</td>
<td>154.01</td>
<td>192.42</td>
</tr>
<tr>
<td>BCLO</td>
<td>15</td>
<td>135.04</td>
<td>74390.95</td>
<td>65.47</td>
<td>148.29</td>
<td>79255.23</td>
</tr>
<tr>
<td>Beta</td>
<td>7</td>
<td>189.52</td>
<td>57.87</td>
<td>64.77</td>
<td>208.16</td>
<td>58.27</td>
</tr>
<tr>
<td>Beta</td>
<td>15</td>
<td>202.64</td>
<td>124.79</td>
<td>63.70</td>
<td>218.91</td>
<td>121.53</td>
</tr>
<tr>
<td>Beta</td>
<td>31</td>
<td>181.14</td>
<td>221.92</td>
<td>63.64</td>
<td>208.22</td>
<td>221.83</td>
</tr>
<tr>
<td>Beta</td>
<td>63</td>
<td>176.24</td>
<td>477.23</td>
<td>66.74</td>
<td>193.03</td>
<td>466.03</td>
</tr>
<tr>
<td>Uniform</td>
<td>7</td>
<td>167.66</td>
<td>42.61</td>
<td>64.64</td>
<td>182.27</td>
<td>42.92</td>
</tr>
<tr>
<td>Uniform</td>
<td>15</td>
<td>166.98</td>
<td>83.40</td>
<td>66.29</td>
<td>176.14</td>
<td>82.53</td>
</tr>
<tr>
<td>Uniform</td>
<td>31</td>
<td>162.11</td>
<td>179.92</td>
<td>63.89</td>
<td>176.53</td>
<td>180.52</td>
</tr>
<tr>
<td>Uniform</td>
<td>63</td>
<td>156.53</td>
<td>409.13</td>
<td>63.91</td>
<td>173.57</td>
<td>412.79</td>
</tr>
<tr>
<td>Uniform</td>
<td>127</td>
<td>162.17</td>
<td>1237.34</td>
<td>65.30</td>
<td>170.74</td>
<td>1232.19</td>
</tr>
</tbody>
</table>

Figure 8.1: Encryption algorithm initialisation times
Figure 8.2: Average encryption times

![Graph showing average encryption times with different algorithms.]

Figure 8.3: Decryption algorithm initialisation times

![Graph showing decryption algorithm initialisation times with different algorithms.]

RAM. The VMs were hosted in a heterogeneous OpenNebula cloud. In addition, a secure Linux VM having 2 vCPUs and 8 GB RAM was used to generate/encrypt and decrypt/verify the data.

Our implementation is pure, unoptimised Java utilising the JScience library [103] arbitrary precision integer classes. It is denoted as algorithm \textit{GACD} in Table 8.1 and Figures 8.1 to 8.4. In addition, to provide comparison for our algorithm we have implemented Boldyreva et al.'s algorithm (referred to as \textit{BCLO}) [53] along
8.4. EXPERIMENTAL RESULTS

Figure 8.4: Average decryption times

with two variants of the Boldyreva et al. algorithm. These latter variants are based on our generic version of Boldyreva et al.’s algorithm (see section 8.3.1). One is an approximation of Boldyreva et al.’s algorithm which samples ciphertext values from the Beta distribution (referred to as Beta in Table 8.1). The derivation of this approximation is given in section 8.3.2. The second samples ciphertexts from the uniform distribution (referred to as Uniform in Table 8.1). This variant appears in Popa et al.’s CryptDB [272] source code [273] as ope-exp.cc. The mean timings for each experimental configuration is tabulated in Table 8.1. The chosen values of \( \rho \) for each experimental configuration are as a result of the implementations of Boldyreva et al. and the Beta distribution version of the generic Boldyreva algorithm. The Apache Commons Math [16] implementations of the hypergeometric and Beta distributions we used only support Java signed integer and signed double precision floating point parameters respectively, which account for the configurations seen in Table 8.1. To provide fair comparison, we have used similar configurations throughout. It should be pointed out that, for the BCLO, Beta and Uniform algorithms, when \( \rho = 7 \), this will result in only 128 possible ciphertexts, even though we have 10,000 inputs. This is because these algorithms will only encrypt each plaintext to a unique value. Such a limited ciphertext space makes these algorithms trivial to attack. Our algorithm will produce 10,000 different ciphertexts as a result of the “noise” term. Each ciphertext will have an effective entropy of at least 21 bits for \( \rho = 7 \) (see section
So, our algorithm is more secure than BCLO, Beta, and Uniform for low entropy inputs.

As shown by Table 8.1, our work compares very favourably with the other schemes. The encryption times of our algorithm outperform the next best algorithm (Uniform) by factors of 28 ($\rho = 7$) to 520 ($\rho = 127$). Furthermore, the decryption times grow sublinearly in the bit length of the inputs. Compare this with the encryption and decryption times for the generic Boldyreva algorithms which, as expected, grow linearly in the bit length of the inputs. Boldyreva et al.’s version performs even worse. We believe this is down to the design of the algorithm, as stated in [53], which executes $n$ recursions where $n$ is the bit-size of the ciphertexts. We also discovered that the termination conditions of their algorithm can result in more recursions than necessary.

It should also be noted that the size of the ciphertext generated by each algorithm seems to have minimal bearing on the MR job execution time. Table 8.1 shows that the job timings are similar regardless of algorithm.

Of course, it is impossible to compare the security of these systems experimentally, since this would involve simulating unknown attacks. But we have shown above that the GACD approach gives a better theoretical guarantee of security than those of [52, 53, 312], which define security based on a game, rather than on the conjectured hardness of a known computational problem.

### 8.4.1 Microsoft Azure Cloud

As in the previous chapter, we also scaled our experiment to a large HDInsight cluster in Microsoft’s Azure cloud. The experimental setup was identical to that presented in section 7.7.1. Again, as a result of the large number of inputs (106,272,000), the input data was generated and encrypted using MapReduce programs running on the HDInsight cluster. In addition, we also had a MapReduce program to decrypt the data and verify that it had been correctly sorted.

For this experiment, we only performed the tests for our own OPE algorithm (GACD) and the variant of Boldyreva et al.’s algorithm sampling from the uniform distribution (Uniform). This was as a result of unresolved issues with converting the encryption program to a MapReduce job for the Beta variant of
Table 8.2: Timings for each experimental configuration \( (n = 106272000) \). \( \rho \) denotes the bit length of the unencrypted inputs. \( \text{Init} \) is the initialisation time for the encryption algorithm, \( \text{Enc} \) is the mean time to encrypt a single integer, \( \text{Exec} \) is the MR job execution time, \( \text{Dec} \) is the mean time to decrypt a single integer.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>( \rho )</th>
<th>Encryption Init. (ms)</th>
<th>Encryption Enc. (( \mu s ))</th>
<th>MR Job Exec. (s)</th>
<th>Dec. (( \mu s ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>GACD</td>
<td>15</td>
<td>96.74</td>
<td>6.97</td>
<td>59.97</td>
<td>4.32</td>
</tr>
<tr>
<td>GACD</td>
<td>31</td>
<td>93.9</td>
<td>7.95</td>
<td>63.02</td>
<td>4.58</td>
</tr>
<tr>
<td>GACD</td>
<td>63</td>
<td>124.4</td>
<td>8.74</td>
<td>71.76</td>
<td>7.28</td>
</tr>
<tr>
<td>GACD</td>
<td>127</td>
<td>128.88</td>
<td>9.93</td>
<td>92.09</td>
<td>10.31</td>
</tr>
<tr>
<td>Uniform</td>
<td>15</td>
<td>71.24</td>
<td>307.71</td>
<td>56.53</td>
<td>280.22</td>
</tr>
<tr>
<td>Uniform</td>
<td>31</td>
<td>77.67</td>
<td>498.78</td>
<td>54.89</td>
<td>506.35</td>
</tr>
<tr>
<td>Uniform</td>
<td>63</td>
<td>66.74</td>
<td>1248.96</td>
<td>59.4</td>
<td>1324.25</td>
</tr>
<tr>
<td>Uniform</td>
<td>127</td>
<td>68.35</td>
<td>4018.86</td>
<td>69.02</td>
<td>4360.11</td>
</tr>
</tbody>
</table>

Figure 8.5: Average encryption times

Boldyreva et al.’s algorithm.

As one can see from Table 8.2 and Figures 8.5 and 8.6, the magnitude of the difference in performance between the two algorithms remains the same. In comparison with the results presented previously in this chapter, we note that, in both cases, the time to encrypt has increased approximately threefold and the time to decrypt twofold (GACD) and threefold (Uniform). This increase may be as a result of memory contention between map tasks running on the same worker.
node, particularly since the encryption process is memory intensive as a result of using arbitrary precision integers.

8.5 Conclusion

Our work has produced an OPE scheme based on the general approximate common divisor problem (GACDP). This appears to be the first OPE scheme to be based on a computational hardness primitive, rather than a security game. We have described and discussed the scheme, and proved its security properties, in section 8.2. In section 8.4 we have reported on experiments to evaluate its practical efficacy, and compare this with the scheme of [53]. Our results show that our scheme is very efficient, since there are $O(1)$ arithmetic operations for encryption and decryption. As a trade-off against the time complexity of our algorithms, our scheme produces larger ciphertexts, $\sim 3.67$ times the number of bits of the plaintext. However, as pointed out in section 8.4, ciphertext sizes had minimal impact on the running time of the MR job used in our experiments.

With regard to our stated purpose, our experimental results show that the efficiency of our scheme makes it suitable for practical computations in the cloud.
We have noted that, like any “true” OPE, our scheme cannot guarantee indistinguishability under CPA \cite{53}, unlike the non-OPE protocols of Popa and others \cite{189,271}. However, with proper choice of parameters, we believe that its security is strong enough for the purpose for which it is intended: outsourcing of computation to the cloud.

To conclude, our OPE scheme implements the OPE scheme required by CryptMR. Furthermore, it significantly outperforms the Boldyreva et al. scheme \cite{53} implemented in related work \cite{272,311,313}. 
Chapter 9

Symmetric Searchable Encryption for CryptMR

9.1 Introduction

In this chapter we present details of our searchable symmetric encryption (SSE) scheme. This scheme implements the SSE scheme required by CryptMR. Our scheme is a generic SSE intended for a broad range of applications. It should be noted that more efficient constructions, such as an searchable encrypted index, can be built using our SSE scheme as a primitive to better support a particular application. We discuss this in the chapter conclusion (9.4).

In this section, we specify changes to the formal model given in section 6.2.2. We also note any further observations as a result of these changes (9.1.2). Additionally, we present a short discussion of related work (9.1.3). In section 9.2, we present our scheme and discuss its security. In section 9.3, we present results of experimental work conducted to validate our algorithms. Finally, in section 9.4 we conclude the chapter.

9.1.1 Formal Model of Scenario

To add to the model of section 6.2.2 we note that the function \( f \) we wish to compute is that which returns those inputs that match a particular pattern.
This pattern is encrypted, and is input to the search function computed in the cloud environment. The results of the computation are the encryptions that have been successfully matched.

Although our primary application model is single-party secure computation, for our SSE scheme we can extend the model to a multiparty scenario. In this new scenario, we have four actors: a data owner, a search broker, a search client and a service provider. The data owner encrypts the data \( m_1, \ldots, m_n \) using the SSE scheme and uploads it to the (cloud) service provider. The data owner uploads a query function, \( f \) to the search broker, a trusted party. \( f \) returns an encrypted query \( q \) on a plaintext query input \( m \). The search client submits the plaintext query \( m \) to the search broker. The search broker uses \( f \) to generate an encrypted query \( q \). \( q \) is submitted to the service provider which returns its results \( c_i, i \in R \subseteq [1, n] \) to the search broker. The search broker forwards the results to the search client.

### 9.1.2 Further Observations from Scenario

For our multiparty scenario, we observe that the search broker must be a trusted party as it is able to observe plaintexts and their matched ciphertexts. We also note that the client cannot decrypt ciphertexts unless it is supplied the secret key \( sk \) by the data owner. However, there may be scenarios, such as computation of statistics where the client only needs the number of results returned rather than the plaintexts.

### 9.1.3 Related Work

Our scheme is based on the scheme of Song et al. [309]. However, it addresses some impracticalities of that scheme. Song et al’s scheme relies on the use of a stream cipher to apply randomness to a deterministic encryption of the plaintext. A bit stream and a MAC generated from the bit stream are XORed with the deterministic ciphertext. However, this system is stateful. It requires the ciphertexts to arrive in the same order for decryption that the plaintexts were in when encrypted. Our scheme is stateless.

Our scheme has some similarities to the first scheme presented in [8]. This scheme
encrypts the plaintext and computes a MAC of the plaintext and outputs the concatenation of the two as ciphertext ("encrypt-and-MAC"). However, as the MAC algorithm is deterministic, ciphertexts of the same plaintext are distinguishable as a result of the MAC component. This makes Amanatidis et al. [8] no more secure than a deterministic encryption scheme. Our scheme is "encrypt-then-MAC". As a result, two ciphertexts of the same plaintext will be different, including the MAC component.

Our work also has similarities to message-locked encryption [31]. Their scheme also uses "encrypt-then-MAC". However, in their scheme, the encryption key is a function of the message. This means that encryptions of the same plaintext will be identical. For their stated purpose of removal of duplicates in a database, this is a desirable property as the ciphertexts preserve the equality on plaintexts. However, as noted by [132], such a system leaks substantial information about plaintexts. Our scheme does not leak the equality of plaintexts until a search has occurred, and then, only for plaintexts that match the particular search term.

While the goal of our work is to present a scheme which is generic and can support a variety of applications, we note that elements of the scheme could be used as a "trapdoor" in construction of an encrypted index for more efficient document searching. Goh [144] first demonstrated how to construct such an index using a Bloom filter [48]. Other works improving on Goh are surveyed in [60].

9.2 Searchable Symmetric Encryption (SSE)

Let \( \mathcal{SE} = (K_E, E, D) \) be a symmetric encryption scheme. \( \mathcal{MAC} = (K_M, M, V) \) be a message authentication code scheme. Let \( f: \mathcal{M} \rightarrow K_M \) be a pseudorandom function (PRF), where \( K_M \) is the range of \( K_M \). Then \( \mathcal{SSE} \) is the system of algorithms (\( \text{KGen}, \text{Enc}, \text{Dec}, \text{SearchKGen}, \text{Match} \)) described below.

**Key Generation**

The key generation algorithm, \( KGen \), is detailed in Algorithm 9.1.
9.2. SEARCHABLE SYMMETRIC ENCRYPTION (SSE)

Algorithm 9.1: Key generation algorithm

| Input  | λ: security parameter |
| Output | k_E: the symmetric key, k_E ∈ K_E |
| k_E ← | K_E(λ); |
| 2 return k_E; |

Search Key Generation

The search key generation algorithm, SearchKGen, is defined in Algorithm 9.2.

Algorithm 9.2: Search key generation algorithm

| Input  | m: plaintext, m ∈ M |
| Output | k_M: the search key, k_M ∈ K_M |
| k_M ← | f(m); |
| 2 return k_M; |

Encryption

The encryption algorithm, Enc, is expressed in Algorithm 9.3.

Algorithm 9.3: Encryption algorithm

| Input  | m: plaintext, m ∈ M |
| Input  | k_E: the symmetric key, k_E ∈ K_E |
| Output | c∥t: ciphertext, c ∈ C, t ∈ T |
| c ← | E(m, k_E); |
| k_M ← | SearchKGen(m); |
| t ← | M(c, k_M); |
| 4 return c∥t; |

Decryption

The decryption algorithm, Dec, is expressed in Algorithm 9.4.

Algorithm 9.4: Decryption algorithm

Search Matching

The search matching algorithm, Match, is expressed in Algorithm 9.5.

Algorithm 9.5: Search matching algorithm
Algorithm 9.4: Decryption algorithm

Input : $c \parallel t$: ciphertext, $c \in C, t \in T$
Input : $k_E$: the symmetric key, $k_E \in K_E$
Output: $m$: plaintext, $m \in M$

1. Retrieve $c$ from $c \parallel t$;
2. $m \leftarrow D(c, k_E)$;
3. return $m$;

Algorithm 9.5: Search matching algorithm

Input : $c \parallel t$: ciphertext, $c \in C, t \in T$
Input : $k_M$: the search key, $k_M \in K_M$
Output: True or False

1. Retrieve $c$ and $t$ from $c \parallel t$;
2. $t' \leftarrow M(c, k_M)$;
3. if $t = t'$ then return True;
4. else return False;

9.2.1 Security

Let $\lambda$ be the security parameter.

We cannot expect SSE to be IND-CPA secure, since an adversary with access to an encryption oracle can distinguish between ciphertexts $c_0, c_1$ of known plaintexts $m_0, m_1$ by computing $f(m_0), f(m_1)$ and matching their tags. But we can ask for a weaker condition, that the adversary knows nothing about the plaintexts for ciphertexts with tags that have not been matched. We will try to capture this notion in the following security game.

1. The challenger $C$ chooses a security parameter $\lambda$ and uses it to generate a key $k_E \in [1, 2^\lambda]$, and a pseudorandom function $f : \mathcal{M} \rightarrow \mathcal{K}_M = [1, 2^\lambda]$.

2. The adversary $A$ chooses any multiset of plaintexts $Q$, where $n = |Q|$ is polynomial in $\lambda$. C returns the corresponding ciphertext, tag pairs $c \parallel t$ for $m \in Q$. A obtains $f(m)$ for all $m \in Q$, but not for any $m \notin Q$.

3. $A$ chooses two further plaintexts $m_0, m_1 \notin Q$. Before choosing $A$ may then carry out any polynomial time computations on the triples $m, c(m), f(m)$ ($m \in Q$), including computing their tags.

4. $C$ chooses a bit $b \in \{0, 1\}$ and returns the ciphertext, tag pair $c_b \parallel t_b$ for the
plaintext \( m_b \).

5. A chooses a bit \( d \in \{0, 1\} \) without carrying out any further computations.

6. A wins if \( d = b \), otherwise C wins.

Let the adversary’s optimal probability of winning be \( \frac{1}{2} + \varepsilon \). Clearly \( 0 \leq \varepsilon \leq \frac{1}{2} \), since the adversary can obtain \( \varepsilon = 0 \) by choosing \( d \) to be a random bit. The probability \( \varepsilon \) is the adversary’s advantage. If \( \varepsilon \) is negligible, we will say that \( \text{SSE} \) is unreadable plaintext indistinguishable under chosen plaintext attack (UP-IND-CPA).

**Theorem 16.** If \( \text{SE} \) is IND-CPA secure, \( f \) is a secure pseudorandom function, then \( \text{SSE} \) is UP-IND-CPA.

Clearly this scheme becomes vulnerable once matching has occurred. Given enough observed matches, an adversary could record the search keys and the number of ciphertexts returned and use this information to deduce the plaintexts. Therefore, while this system will tolerate a small number of matches, eventually the risk of disclosure becomes large.

### 9.2.2 False Positives

We now show that the probability that a search returns a false positive, that is, it returns a ciphertext which is not an encryption of the plaintext search term, is negligible, under reasonable assumptions about \( \text{SE} \) and \( \text{MAC} \). We require the \( \text{MAC} \) to satisfy a “uniqueness” condition on tags of the form that

\[
\Pr \left[ \exists m, m' \in \mathcal{M} : M(c(m), f(m)) = M(c(m'), f(m')) \right]
\]

is negligible. The probabilities are over the (pseudo)randomness in \( E \) and \( f \).

**Theorem 17.** Let \( E \) be a sufficiently random block cipher with \( B = |\mathcal{C}| \). Let \( D \) be the size of the dictionary of valid \( m \in \mathcal{M} \), let \( T = |\mathcal{T}| \) and let \( c_m = E(k_E, m) \). Then

\[
\Pr \left[ \exists m, m' \in \mathcal{M}, m \neq m' : M(c_m, f(m)) = M(c_{m'}, f(m')) \right] \leq \frac{1}{BT} + \left( 1 - \frac{1}{B} \right) \frac{D^2}{2T}.
\]
If $B$ is sufficiently large and $D^2 \ll T$, this probability is negligible.

We implement SE using AES-256 in CBC mode with a random IV. We implement $f$ using AES-256 in CBC mode with a constant IV. Finally we implement MAC as CBC-MAC using AES as the underlying block cipher. AES has a block size of 128 bits and CBC-MAC generates tags of half the block cipher’s block size. Therefore, we have $B \approx 2^{128}$ and $T = 2^{64}$. If we assume that we are enciphering words from the English language, then there are approximately 180,000 words in current use, so $D = 180,000$. Using Theorem 17, we estimate the probability of any false positive to be approximately $1.75 \times 10^{-9}$.

\section*{9.3 Experimental Results}

We evaluated our algorithm using a variant of the canonical word count MapReduce (MR) program. Our variant counted the number of occurrences in a file that matched a particular keyword.

Our experimental setup consisted of a secure client, a Linux VM with 1 vCPU and 8GB RAM, and a Hadoop cluster of 17 nodes, one master and 16 slaves, implemented as Linux virtual machines in a heterogenous OpenNebula cloud. Each VM had 1 vCPU and 2GB RAM. For our source text, we used a fragment of the Wikipedia corpus, consisting of 47,200 pages. The file was preprocessed by removing the XML tags and punctuation and converting all words to lower case.

Four algorithms were implemented and evaluated: our own (designated SSE), a deterministic encryption scheme using AES-256 in CBC mode with a constant initialisation vector (designated DET), the Song et al. scheme \[309\] (designated SWP), and the Amanatidis et al. scheme \[8\] (designated ABO). We encrypted the source text using each algorithm on the secure client and then executed the version of our word count programme pertinent to that encryption scheme in the cloud. Finally, we decrypted the MR job output on our secure client. The timings for encryption, decryption, and the MR job are given in Table \[9.3\] and Figure \[9.1\].

We can see that DET and SWP show the best performance for encryption. As expected the DET algorithm is fastest as it simply performs a deterministic encryption of the word. Our SSE algorithm and ABO were both implemented so
9.3. **EXPERIMENTAL RESULTS**

Table 9.1: Timings for Evaluated SSE Algorithms. \textit{Alg} denotes the algorithm used, \textit{Enc. Init.} is the initialisation time for the encryption algorithm, \textit{Avg. Enc.} is the average time to encrypt a word, \textit{Exec.} is the time to execute the MapReduce job, \textit{Dec. Init.} is the initialisation time for the decryption algorithm, \textit{Avg. Dec.} is the average time to decrypt a word.

\begin{tabular}{|l|c|c|c|c|c|}
\hline
\hline
ABO & 637 & 23.47 & 120.63 & 521.67 & 32.49 \\
DET & 311.21 & 0.42 & 27.43 & 336.33 & 141.89 \\
SSE & 577.78 & 21.73 & 134.19 & 545.26 & 24.88 \\
SWP & 501.67 & 3.15 & 31.35 & 552.33 & 140.67 \\
\hline
\end{tabular}

Figure 9.1: Timings for Evaluated SSE Algorithms

that the output of the encryption algorithm is a JSON \cite{123} string consisting of the Base64 \cite{176} encodings of the ciphertext and CBC-MAC tag. Clearly, the overhead for the building the JSON object and then serialising it, is significant because the SWP algorithm is also performing encryption and then construction of a CBC-MAC tag. This may be as a result of the \texttt{javax.json} Java library \cite{257} we used for JSON encoding. For the job execution time, we see a similar slow down for the ABO and SSE algorithms which we must attribute to parsing the serialisation of the ciphertext and tag as a JSON object. The ABO algorithm is simply matching tags, and performs less operations than SSE and SWP as part of the matching process, so it should be faster than SWP. For decryption, SSE and ABO fare best because each algorithm stores the tag used to search over. This means that the output of the MR job can simply be validated by checking that the correct tag has been returned. However, for the DET and SWP algorithms
we must decrypt the returned ciphertext to validate that we have produced the correct results.

In conclusion, we can see that implementing the ABO and SSE algorithms so that we do not use JSON would improve performance so that it is on par with the SWP algorithm.

9.4 Conclusion

In this chapter, we presented details of a novel SSE scheme inspired by the Song et al. algorithm \cite{309} which implements the CryptMR SSE scheme. Our scheme uses the “encrypt-then-MAC” paradigm as the basis of its security and searchability. It has the advantage that encryption is stateless, meaning that, unlike the Song et al. algorithm, the order that items arrive in for encryption or decryption is irrelevant. With regard to MapReduce, after the map, shuffle and reduce phases we cannot guarantee that the output data will be in the same order as the input data. Therefore, without some saved state, Song et al.’s scheme would be unable to decrypt the results unlike our scheme. Furthermore, our scheme is proven to be secure against chosen plaintext attack provided no searching has occurred, as is the case for the Song et al. scheme.

We have conducted extensive evaluation of our algorithm and, while they are not as emphatic as the experimental results of previous chapters, they indicate how we might optimise our implementation for significantly better performance. We intend to make those changes and present the revised experimental results in future work.

With regard to our algorithm, as we stated in the introduction to this chapter, it was our intention to provide a generic primitive to allow for construction of MapReduce programs over searchable encrypted data. We recognise that an inverted index based scheme would be more efficient to search over. However, we believe that our scheme could form the basis of such a construction. We may investigate this in future work.
Chapter 10

A Result Verification Method for MapReduce

This chapter presents a method to provide checking of intermediate results of mappers and final results of reducers by sampling results and checking their values against pre-computed results. Such a system is important for CryptMR because while computing on encrypted data protects the data from snooping it does not protect the data from tampering. Our system gives a strong probability that all results are genuine and have not been altered. Unlike similar work \cite{39, 41, 235, 282, 337, 352}, we provide a system which requires sampling of fewer results.

10.1 Introduction

A major weakness of the MapReduce model is that if data computed by one mapper or reducer is incorrect then the whole computation will provide an incorrect result. With reliable computing nodes, and where security is not a concern, this is not really a problem. However, if a malicious individual deliberately tries to disrupt the computation this becomes a serious flaw. Therefore, in order to check that mapper and reducer computations have been completed correctly, we require some means of verifying their results. For secure computation in the cloud we have two further challenges regarding data verification. Suppose we use message authentication codes (MACs) or digital signatures to ensure data has not been tampered with. First, we have a problem regarding distribution of secret keys
used in the MAC or digital signature algorithms across the MapReduce cluster. Not only must the secrets be securely distributed to cluster nodes, they must also be securely stored on the cluster nodes, such that workers are able to sign the data they generate but the secret is not exposed if the cluster node is compromised. Secondly, we require a series of MACs or signatures to verify that the data was not tampered with at any step in the computation. A secure hardware solution, such as SGX [11], might solve the first problem, allowing for the MAC or signature to be computed in a secure enclave on the machine. For the second problem, we might use a Merkle tree [223, 225] in a similar fashion to the work of [84]. However, using SGX as a solution first requires that the processors on cluster nodes support SGX. Secondly, SGX support for virtual machines is currently in a preliminary form [170]. With regard to the use of Merkle trees, such a solution requires extensive computation to generate and verify the signatures.

Therefore, it would be desirable to have a lightweight result verification system that is portable to a wide variety of MapReduce implementations and hardware architectures. The simplest method would be to use redundancy. Many mappers would be assigned the same input split and compute their results. We would decide the correct output by majority decision. The outputs that are the same for the largest number of mappers would be taken as correct. Verification of reducers would proceed similarly. However, this method requires a considerable amount of duplication of work.

To reduce the amount of replication of computation we can randomly duplicate some of the work of mappers or reducers on a trusted node, such as a machine in a private cloud. To escape detection, an attacker must correctly choose which mappers or reducers will be sampled. Furthermore, the attacker must choose which inputs to the map or reduce function will be used for verification purposes. If he tampers with the results of those mappers or reducers whose computations are duplicated by the verifier it will be detected. [352] and [355] present schemes which use this approach.

10.1.1 Terminology

In addition to the MapReduce terminology introduced in chapters 2 and 4, we add further terms to describe characteristics of the raw input data. In raw data, we
find that the data is divided into items which are demarcated by some separator, such as whitespace, a punctuation mark or a particular bit pattern. A comma separated values (CSV) file typically separates data items using commas. We refer to the data items in the raw data as fields. Furthermore, a group of data items may be further demarcated from other data by another separator, such as a line feed or carriage return character. In this way, data items are collected together into a mutually intelligible group of items, e.g. a line of a CSV file. We refer to this group of items as a record. For example, the weather station CSV data shown in Table 10.1 contains 76 rows. Each row is a record. Furthermore, the data on each row is separated into several columns. Each column is a field.

### 10.2 Assumptions

The design of our verification system is based on the following assumptions:

1. The MapReduce master, the verification system and the distributed file system (DFS) are trustworthy.

2. The workers (mappers and reducers) are not trustworthy.

3. Communication between all nodes is trusted.

### 10.3 Adversary Model

We assume that any adversary is actively and maliciously trying to falsify the intermediate and final results for the aim of invalidating the computation. This adversary may have control of nodes in the MapReduce cluster, and thereby have control of mappers and reducers. We assume that a malicious adversary has the capacity to overwrite genuine results with falsified results or may alter the map and reduce user-defined functions so that falsified results are produced.

Therefore we can categorise maliciously controlled mappers and reducers as belonging to one of three classes: lazy cheaters, hoarding cheaters, and malicious cheaters. A lazy cheater is one, where rather than computing $f$ on a piece of data, it computes $\hat{f}$, where $\hat{f}$ is a much easier function to compute. For example, $\hat{f}$
## Table 10.1: Sample Meteorological Office weather station observation data for 12:00am 18/7/2015

<table>
<thead>
<tr>
<th>Site Code</th>
<th>Site Name</th>
<th>Latitude</th>
<th>Longitude</th>
<th>Region</th>
<th>Observation Time</th>
<th>Observation Date</th>
<th>Wind Direction</th>
<th>Wind Speed</th>
<th>Wind Gust</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>SHOREHAM (3876)</td>
<td>50.8360</td>
<td>-0.2920</td>
<td>London &amp; South East England</td>
<td>00:00</td>
<td>2015-07-18</td>
<td>W</td>
<td>6</td>
<td>3876</td>
</tr>
</tbody>
</table>
might be the identity function. A hoarding cheater is one where the cheater withholds publishing results, particularly where a single result might be considered valuable. A malicious cheater is one who actively and deliberately falsifies results.

In this chapter, we assume our adversarial mappers and reducers are mainly of the lazy and malicious cheater types. Hoarding cheaters may be considered as similar to a mapper or reducer that is not responsive for any other reason, e.g., hardware or software failure. However, in the case of a hoarding cheater, it may try to release the results it withheld to a third party at a later time. Therefore, we can address hoarding cheaters by marking the process as failed, quarantining the node on the network, and re-scheduling the work on another node.

10.4 Core Idea

A MapReduce input split is comprised as a number of records. These records are parsed as key-value pairs by the input reader. For example, the records in the sample data in Table 10.1 might be parsed so that the “Site Code” field is the key of the pair and that the remaining fields are stored, as a comma-separated list, as the value of the pair. In MapReduce applications, there is a one-to-one correspondence between each input record and each key-value pair consumed by a mapper. Therefore, there is a one-to-one relationship between the list of
pairs output by each application of the \textit{map} function and each record in the input split. We can use this relationship as the basis of a verification method for mapper results.

Suppose that we choose one record from an input split. This record can be parsed as a key-value pair and then have the \textit{map} function applied to it. This function application will produce a list of key-value pairs. We can then store these results for comparison with the results produced by the mapper. We require that we are able to identify which record in the input split we chose. We can number each record consecutively in the order which they appear in the split and use these numbers as indices to each record.

As a mapper processes an input split, we record the output of each \textit{map} function application. These function results are stored in the same order as the corresponding records in the input split. When the mapper has completed applying the \textit{map} function to the last key-value pair derived from the input split, we are then able to verify its output correctness by comparing the results it has generated with those that we have generated. In the total set of results generated by the mapper, we locate the subset of results that are the output of the map function applied to the key-value pair that corresponds to the record we chose. If the subset of results generated by the mapper match those that we have generated, then we have some assurance that the mapper is behaving correctly and that it has not been compromised.

In order to be completely sure that a mapper is computing results in a trustworthy manner, we could process every record in the input split, generate a set of results and compare those with the results generated by the mapper. However, this requires duplicating the computation required to process the input split. It would be more desirable to be able to choose a small number of records from the input split and use these records for validation. In particular, we would like the number of records sampled for verification purposes to be much smaller than the total number of records in the input. Furthermore, we would demand that the records chosen for verification purposes should not be easily identified by any attacker. If this was not the case, it becomes trivial for a malicious mapper to submit a set of results for verification that matches those generated by ourselves while writing false results to its local storage. Therefore, we choose a record or records at random from the set of records in the input split. We proceed as outlined
10.4. CORE IDEA

A malicious mapper now has to correctly guess which records have been chosen for verification purposes if it wishes to submit falsified results without being detected. The probability that it successfully does so is:

\[
\frac{n}{R} \times \frac{n-1}{R-1} \times \ldots \times \frac{1}{R-n+1} = \frac{n!(R-n)!}{R!} = \binom{R}{n}^{-1}
\]

where \( R \) is the number of records in the input split and \( n \) is the number of records chosen for verification. Where \( R \gg 1 \), the probability that the malicious mapper correctly guesses the record(s) chosen for verification is very small. Suppose, for example, that \( R = 1,000 \) and \( n = 50 \) (5% of records), this probability is \( 1.06 \times 10^{-85} \).

We use the method described above as the basis of our trusted result verification system. Our result verification system takes our place as shown in Figure [10.1] in that it randomly chooses records from the input split. The identity of these records is known only to the verification system. It parses each record as a key-value pair, and compute the results of the map function. In order to reduce the time required to compare the verification system’s results and the mapper’s results, we apply a hash function to the set of results generated by the verification system. The set of key-value pairs output by the map function is supplied as input to a hash function and a tag is generated. The authentication system stores the triple of index of the input split, the index of the pair selected within the input split and generated tag.

As a mapper processes its input split, it applies the map function to each key-value pair parsed and generates the output key-value pair set for each application. The hash function is applied to each set to generate a tag. The mapper stores the pair of the index of each record from the input split and the tag generated in a file on local storage. When the mapper wishes to commit its results for processing by reducers, it notifies the verification system. The verification system fetches the file containing the verification hashes computed by the mapper. The authentication system locates the pair supplied that corresponds to the index of the record it selected for authentication. It compares the tag supplied by the mapper with the tag it previously computed. If these tags match, the mapper’s results are confirmed as valid. Otherwise, they are confirmed as invalid.

This solution relies on random selection of the records used for verification in
order to provide strong assurance that the results have not been tampered with. A compromised mapper that wishes to submit false results must successfully guess which records have been used for authentication and then compute the correct map results for those records. However, where there are a large number of records in an input split, the probability that it will successfully guess the required records is small. As the submission of the verification results is one-time only, the compromised node is unable to brute force the authentication system by repeating the process over and over until it correctly guesses the record. With high probability, an attacker that submits a false result set will be detected.

As mentioned previously, the records used for verification may be randomly selected from those in the input split or artificial data randomly inserted into the input split. In the latter case, this data is indistinguishable from genuine data. The use of such artificial data does not alter the verification process but it may be used as provenance should the input data be leaked.

10.5 Issues

The core idea for this solution is quite simple yet there are several further issues that need to be detailed and addressed. These issues are detailed below.

10.5.1 Single Verification Server (VS)

If there is only one VS, for example, the master node, then this represents a single point of attack. Once the VS is compromised, the verification system is compromised. If the verification system is distributed across a number of VS’s then compromising the system requires compromise of all the servers. Furthermore, we require the distribution of the system across the servers to be configured such that compromising one of the servers does not allow an attacker to circumvent the authentication system.
10.5.2 Input splits containing small numbers of records

If an input split contains a small number of records then the probability that an attacker correctly guesses the record used for authentication is high. In addition, in order to compute the check values, the authentication system will duplicate a large proportion of the MapReduce computation.

10.5.3 Malicious Partitioner

A mapper must compute the hash of an output key-value pair set before the set is passed to the partitioning function. Otherwise, once the records have been placed into the partition “bins”, there is no correlation between the output records in the bins and the records in the input split. As the partition function may be user defined, this affords an opportunity to alter the data after the checksums have been computed. A malicious partitioner might be constructed to alter the data as it is partitioned. Alternately, an attacker with control of the node’s file system may be able to alter the partitioned data.

10.5.4 Size of check data

The amount of check data that a mapper will need to submit to the VS may prove to be quite large. The total amount of check data sent will be equal to the size of the hash tag multiplied by the number of records in the input split. Typical hash functions generate 128 to 512 bit tags. Where an input split has a large number of records this will result in a large message sent to the VS.

However, reducing the number of tags sent to the VS would alter the probabilities that an attacker is able to successfully guess which record in the input split is used for result authentication. If only the tag for this record is requested, we have announced to an attacker which of the records we are checking. It then proves possible to generate false map results for all records in the input split, yet still apply the map function correctly to the checked record and compute a hash tag which will be accepted. An alternative is to try and disguise which of the records we are checking by asking for a subset of the input records. However, we have now alerted an attacker that the record to be checked is among those requested.
This increases the probability of an attacker correctly guessing this record.

Alternatively, we could reduce the size of the hash tag by using an algorithm which produces smaller tags. However, this would increase the chance of a collision where two pieces of distinct data produce the same tag. If the probability of a collision is close to the probability that an attacker can correctly guess the records for verification, then the system is flawed.

10.6 High Level Description of the Verification Process

In this section we provide a high-level description of the MapReduce result verification process. The verification process modifies the MapReduce computational structure so that at the end of the map and reduce phases respectively, the MapReduce computation enters a commitment phase. In this commitment phase, the result verification infrastructure takes over and computes whether the results produced by the mappers and reducers are reliable or not. If the results have been deemed reliable then they are committed and the MapReduce computation resumes. If they are not reliable then the computation will be halted.

The result verification infrastructure consists of two types of daemons, a single verification master and several result verifiers. The verification master is responsible for co-ordinating the verification process. Its job is to aggregate the verification results of the result verifiers, to decide whether the map or reduce phase results should be committed or rejected, and to inform the MapReduce master whether the computation should be resumed or halted. The result verifiers are responsible for duplicating work produced by mappers or reducers as part of the map and reduce phases of the computation respectively. The results of this duplicated work are compared to the results produced by the mapper or reducer. If these results are the same then the mapper or reducer is deemed reliable. If not, then it is deemed unreliable.
10.7 Detailed Description of the Verification Process

In this section we describe the solution. The solution has been modified from the core idea to address most of the issues detailed in section 10.5.

The verification system is distributed across several trusted verification servers. Each server performs the initialisation process as described in section 10.4.

The workers process the data as outlined in 10.4. When they are ready to commit their results, it notifies each verification server. Each verification server compares the hash tag supplied with the tag it has computed for the input record it chose.

10.7.1 Verification of Mapper Results

Let there be $N$ input splits. An input split contains approximately $r$ records. The process is initialised by algorithm 10.1 which is performed by one or more verification servers (VS).

$s_i$ is the $i$th input split. $RecordByteRanges$ is a function which calculates the bytes that correspond to the start and end of each record in the input split. $(b_1, b_2, \ldots, b_n)$ is the list of byte ranges computed by $RecordByteRanges$. $ParseAsKeyValuePair$ is a function which parses the byte range for a single record as a key-value pair. $map$ is the MapReduce application $map$ function. $Check$ is a sparse array which stores the check values used to verify each mapper.

Initialisation Algorithm

Each result verifier (VS) initialises the verification process according to algorithm 10.1. The VS randomly chooses which input splits to verify. It then randomly chooses which records within those input splits to verify. It selects each record, noting its position within the input split and which split it belongs to. Each record is parsed, and the map function is applied to it. The list of key-value pairs emitted is hashed to produce a tag which is stored in the $Check$ array.
Algorithm 10.1: Initialisation algorithm

1. forall \( i \in [1, N], j \in [1, r] \) do
2.     \( \text{Check}_{ij} \leftarrow 0; \)
3. end
4. Randomly choose \( NV \subset [1, N]; \)
5. forall \( i \in NV \) do
6.     \( (b_1, b_2, \ldots, b_r) \leftarrow \text{RecordByteRanges}(s_i); \)
7. end
8. end
9. return \( \text{Check}; \)

Verification Algorithm

Each mapper processes an input split as outlined in algorithm 10.2. The mapper parses the split as a list of key-value pairs \((kv_1, kv_2, \ldots, kv_m)\). The mapper applies the map function to each pair, \(kv_i\). The output of each application is a list of key-value pairs, \((kv'_1, kv'_2, \ldots, kv'_m)\). A check value, \(h_i\) is generated by applying a hash function to the key-value pair list output by each map function application. Each check value generated is stored in the order it was generated. In this fashion, the order of the check values corresponds to the order of the records in the input split. If the application of the map function raises an error, then a plaintext value is stored to signify this.

Algorithm 10.2: Mapper processing algorithm

1. Mapper \( M_i \) processes input split \( s_i; \)
2. \( (b_1, b_2, \ldots, b_r) \leftarrow \text{RecordByteRanges}(s_i); \)
3. forall \( j \in [1, r] \) do
4.     \( kv \leftarrow \text{ParseAsKeyValuePair}(b_j); \)
5.     \( (kv_1, kv_2, \ldots, kv_m) \leftarrow \text{map}(kv); \)
6.     \( h_j \leftarrow \text{Hash}(kv_1 || kv_2 || \ldots || kv_m); \)
7. end
8. return \( (h_1, \ldots, h_r); \)

As detailed in algorithm 10.3 when the \( i \)th mapper has completed processing,
it submits a notification message to the result verifier(s). Each result verifier reads the check values stored on the mapper and compares the $j$th check value received with $\text{Check}_{ij}$, where $\text{Check}_{ij} \neq 0$. If the values match, then the results the mapper has calculated are authenticated.

Algorithm 10.3: Mapper checking algorithm

1. Mapper $M_i$ announces intention to commit results;
2. The VS reads array $(h_1, h_2, \ldots, h_n)$ on $M_i$;
3. if $\text{Check}_{ij} \neq 0 \land h_j = \text{Check}_{ij}$ then return Valid;
4. else return Invalid;

The result verifiers that have sampled from the input split corresponding to this mapper compare check results. If all applicable result verifiers return the valid result, then the verification master accepts the mapper’s results. If one result verifier returns the invalid result, then the mapper’s results are rejected and the task is rescheduled on another node.

10.7.2 Verification of Reducer Results

As previously stated, the mappers partition the intermediate data they produce into several “bins” according to a partitioning function common to each mapper. Most commonly, they are partitioned by a hash function.

Each reducer is assigned a partition to process. In the “shuffle” phase of MapReduce each reducer gathers the data from each mapper from the pertinent partition. This data is then sorted by key. Pairs that have the same key are aggregated into a single key-value pair.

Let there be $P$ partitions of the intermediate data generated by each mapper. Once sorted and shuffled the partition contains approximately $K$ key-value pairs. The process is initialised by algorithm 10.4 which is performed by one or more result verifiers (VS).

$p_i$ is the $i$th partition. $\text{ShuffleAndSort}$ is the MapReduce function which gathers, sorts and aggregates the intermediate data. $\text{Check}'$ is a sparse array which stores the check values used to verify each reducer.
CHAPTER 10. RESULT VERIFICATION

Initialisation Algorithm

Each result verifier initialises the verification process according to algorithm 10.4. The result verifier randomly chooses which partitions to verify. It gathers, sorts and combines the pairs for each pair. It then randomly chooses which key-value pairs within those partitions to verify. It selects each pair, noting its position within the partition and which partitions it belongs to. The reduce function is applied to the pair. The list of key-value pairs emitted is hashed to produce a tag which is stored in the $Check'$ array.

Algorithm 10.4: Initialisation algorithm

1. forall $i \in [1, N], j \in [1, r]$ do
2.   $Check_{ij} \leftarrow 0$;
3. end
4. Randomly choose $PV \subset [1, P]$;
5. forall $i \in PV$ do
6.   $(kv_1, kv_2, \ldots, kv_K) \leftarrow ShuffleAndSort(p_i)$;
7.   Randomly choose $SK \subset [1, K]$;
8.   forall $j \in SK$ do
9.     $(kv_1', kv_2', \ldots, kv_m') \leftarrow reduce(kv_j)$;
10.    $Check'_{ij} \leftarrow Hash((kv_1', kv_2', \ldots, kv_m'))$;
11. end
12. end
13. return $Check'$

Verification Algorithm

Each reducer processes an partition as outlined in algorithm 10.5. The reducer applies the $reduce$ function to each pair, $kv_i$. The output of each application is a list of key-value pairs, $(kv_1', kv_2', \ldots, kv_m')$. A check value, $h_i$ is generated by applying a hash function to the key-value pair list output by each reduce function application. Each check value generated is stored in the order it was generated. In this fashion, the order of the check values corresponds to the order of the sorted key-value pairs in the shuffled and sorted partition. If the application of the reduce function raises an error, then a plaintext value is stored to signify this.

As detailed in algorithm 10.6 when the $i$th reducer has completed processing, it submits a notification message to the result verifiers. The result verifier accesses
10.7. DETAILED DESCRIPTION OF THE VERIFICATION PROCESS

**Algorithm 10.5:** Reducer processing algorithm

1. Reducer $R_i$ processes partition $p_i$;
2. $(kv_1, kv_2, \ldots, kv_K) \leftarrow \text{ShuffleAndSort}(p_i)$;
3. forall $j \in (1, \ldots, K)$ do
   4. $(kv_1, kv_2, \ldots, kv_m) \leftarrow \text{reduce}(kv_j)$;
   5. $h'_j \leftarrow \text{Hash}((kv_1 \| kv_2 \| \ldots \| kv_m))$;
4. end
5. return $(h'_1, \ldots, h'_r)$;

The log of check values on the reducer and compares the $j$th check value with $\text{Check}_{ij}$, where $\text{Check}_{ij} \neq 0$. If the values match, then the results the reducer has calculated are authenticated.

**Algorithm 10.6:** Reducer checking algorithm

1. Reducer $R_i$ announces intention to commit results;
2. The VS reads array $(h'_1, h'_2, \ldots, h'_n)$ on $R_i$;
3. if $\text{Check}_{ij}' \neq 0 \land h'_j = \text{Check}_{ij}'$ then return Valid;
4. else return Invalid;

Again, the reducer’s results are only accepted if all applicable result verifiers concur that the results are valid.

### 10.7.3 Probabilistic Analysis

If all the records are falsified the probability of detection is 1. However, if only a fraction of the records are falsified then what is the probability that we successfully detect tampering?

We will consider the mapper case first. The reducer case is similar. Suppose there are $m_c$ always cheating mappers from a pool of $M$ mappers, and that we choose to verify $m_v$ of the $M$ mappers, choosing $r_i$ records from the $i$th mapper’s input split where $i \in \{1, \ldots, m_v\}$, and each $r_i < N$, the number of records in each input split. In all cases, we assume sampling without replacement, and approximate the probability of choosing a cheater as:

\[
\Pr[\text{Choose a cheater}] = \frac{m_c}{M}
\]
This expression is exact if we assume sampling with replacement.

Therefore, the probability that we successfully detect at least one cheater is:

\[
\Pr[\text{Detect no cheater}] = \left(1 - \frac{m_c}{M}\right)^{m_v}
\]

\[
\implies \Pr[\text{Detect one cheater}] = 1 - \left(1 - \frac{m_c}{M}\right)^{m_v}
\]

As an example, consider the case where \(M = 1000, m_c = 200, m_v = 50\) i.e. 20% of mappers are cheaters and we verify 5% of the mappers. Therefore, the probability we detect cheating is 0.99999.

Now for \(m_v \geq m_c\), the probability that we successfully detect all cheaters is:

\[
\Pr[\text{Detect all cheaters}] = \binom{M-m_c}{m_v} \frac{(M-m_c)!}{m_v!(M-m_v)!} \frac{m_v!(M-m_v)!}{M!}
\]

\[
= \frac{(m_v - m_c)!(M - m_v)!}{(M - m_c)! m_v!} \frac{M!}{(m_v - m_c)!}
\]

\[
= \frac{M(M-1)\ldots(M-m_c+1)}{m_v(m_v-1)\ldots(m_v-m_c+1)}
\]

\[
\approx \binom{m_v}{M} \binom{m_c}{M} \quad (m_v, M \gg m_c)
\]

By way of example, consider the case where \(M = 1000, m_c = 50, m_v = 200\) i.e. 50% of mappers are cheaters and we verify 20% of the mappers. Therefore, the probability we detect all cheaters is:

\[
= \frac{950!}{150!} \times 200! \times 1000!
\]

\[
= 1.40 \times 10^{2418} \times 5.71 \times 10^{202}
\]

\[
= 7.89 \times 10^{374} \times 4.02 \times 10^{2567}
\]

\[
= 4.80 \times 10^{-38}
\]

In this case, the chance of detecting all cheaters is infinitesimally small.

Now assume that the cheaters do not always cheat. They will randomly output a falsified result with probability \(p\), and a correct result with probability \(1 - p\).
Now the probability that the verifier chooses to verify the \( i \)th mapper but does not detect it is cheating given that it is a cheater is:

\[
\Pr[\text{Do not detect}|\text{cheater}] = (1 - p)^{r_i}
\]

Therefore, the probability that the verifier chooses to verify the \( i \)th and does not detect cheating is:

\[
\Pr[\text{Do not detect}] = \frac{m_c}{M} (1 - p)^{r_i}
\]

The probability that we detect one cheater is:

\[
\Pr[\text{Detect one cheater}] = 1 - \prod_{i=1}^{m_v} \frac{m_c}{M} (1 - p)^{r_i} = 1 - \left( \frac{m_c}{M} \right)^{m_v} (1 - p) \sum_{i=1}^{m_v} r_i
\]

As an example consider the case where \( M = 1000, m_c = 200, m_v = 50 \) i.e. 20% of mappers are cheaters and we verify 5% of the mappers. We have \( N = 1000 \) input splits and sample approximately 10% of records in an input split, i.e. each \( r_i \approx 100 \). Each cheating mapper cheats with probability 0.5. Therefore, the probability we detect one cheater is:

\[
\approx 1 - \left( \frac{200}{1000} \right)^{50} (0.5)^{100 \times 50} \\
\approx 1 - 1.12 \times 10^{-35} \times 7.07 \times 10^{-1506} \\
\approx 1 - 7.97 \times 10^{-1541}
\]

The probability that the verifier chooses to verify the \( i \)th mapper and detects it is cheating given that it is a cheater is:

\[
\Pr[\text{Detect cheating}|\text{cheater}] = 1 - (1 - p)^{r_i}
\]

The probability we detect all cheaters given that we choose all cheaters is:

\[
\Pr[\text{Detect cheating}|\text{Choose all cheaters}] = \prod_{i=1}^{m_c} (1 - (1 - p)^{r_i})
\]
Therefore, the probability that we successfully detect all cheaters is:

\[
\Pr[\text{Detect all cheaters}] = \frac{(\frac{M-m_c}{m_v})^m}{M^m} \times \prod_{i=1}^{m_c} (1 - (1 - p)^{r_i}) \\
\approx \left(\frac{m_c}{M}\right)^m \frac{m_v}{m_c} \prod_{i=1}^{m_c} (1 - (1 - p)^{r_i}) \quad (m_v, M \gg m_c)
\]

Again, as an example, consider the case where \(M = 1000, m_c = 50, m_v = 200\) i.e. 5% of mappers are cheaters and we verify 20% of the mappers. We have \(N = 1000\) input splits and sample approximately 10% of records in an input split, i.e. each \(r_i \approx 100\). Each cheating mapper cheats with probability 0.5. Therefore, the probability we detect all cheaters is:

\[
\left(\frac{950}{150}\right) \times \left(\frac{1000}{200}\right) \times (1 - (0.5)^{100})^{50} \\
= 4.80 \times 10^{-38} \times (1 - 7.89 \times 10^{-31})^{50} \\
= 4.80 \times 10^{-38}
\]

Now for the reducer case. The analysis is very similar. Suppose there are \(r_c\) always cheating reducers from a pool of \(R\) mappers, and that we choose to verify \(r_v\) of the \(R\) reducers, choosing \(k_i\) key-value pairs from each reducer’s partition where \(i \in \{1, \ldots, r_v\}\). In all cases, we again assume sampling without replacement, and approximate the probability of choosing a cheater as:

\[
\Pr[\text{Choose a cheater}] = \frac{r_c}{R}
\]

This expression is exact if we assume sampling with replacement.

Therefore, the probability that we successfully detect one cheater is:

\[
\Pr[\text{Detect cheating}] = 1 - \left(1 - \frac{r_c}{R}\right)^{r_v}
\]
Now for $r_v > r_c$, the probability that we successfully detect all cheaters is:

$$\Pr[\text{Detect all cheaters}] = \frac{r_v(r_v - 1) \ldots (r_v - m_c + 1)}{R(R - 1) \ldots (R - r_c + 1)} \approx \left(\frac{r_v}{R}\right)^{r_c} (r_v, R \gg r_c)$$

Now for the case where the cheaters do not always cheat. They will output a falsified result with probability $q$, and a correct result with probability $1 - q$. The probability that we detect cheating is:

$$\Pr[\text{Detect one cheater}] = 1 - \prod_{i=1}^{r_c} \frac{r_v}{R} (1 - q)^{k_i}$$

$$= 1 - \left(\frac{r_v}{R}\right)^{r_c} (1 - q) \sum_{i=1}^{r_c} k_i$$

The probability we detect all cheaters is:

$$\Pr[\text{Detect all cheaters}] = \frac{(R - r_c)}{(r_v - r_c)} \times \prod_{i=1}^{r_c} (1 - (1 - q)^{k_i})$$

$$\approx \left(\frac{r_v}{R}\right)^{r_c} \prod_{i=1}^{r_c} (1 - (1 - q)^{k_i}) (r_v, R \gg r_c)$$

### 10.8 Implementation

As a “proof of concept” we implemented the algorithms described above as several Hadoop MapReduce jobs and Java programs. In order to implement the initialisation algorithms given above, we require a way apply the map and reduce functions to data outside of the Hadoop framework. To achieve this, we implemented two classes `VerifiableMapper` and `VerifiableReducer` which specify static methods for the map and reduce functions. These two classes also implement Algorithms 10.2 and 10.5. User-defined mapper and reducer classes must inherit from these classes.

The verification process consists of the following sequence of programs:

**ReadRecords** This map-only MR job parses the raw input data as key-value
pairs. Because it is a map-only job, the ordering of the key-value pairs output preserves the ordering of the records in the original input split.

**SelectRecords** This program implements Algorithm 10.1 and runs on a secure verification machine. It samples key-value pairs from the output of *ReadRecords*. Each key-value pair has the map function applied to it. The function output is concatenated and hashed using the MD5 algorithm. This is written to a master check file which holds records containing the index of the file sampled (the numerical suffix after `part-m-` in the filename), line number of the pair sampled from the file, and the hash generated.

**RunMappers** This map-only job executes the map phase on the output generated by *ReadRecords*. As each input key-value pair is processed by an individual mapper, check data is written to a file. Each record in the file consists of the line number of the pair in the input file and a hash generated as specified above.

**VerifyTasks** This program implements Algorithm 10.3. The check data generated by each mapper is downloaded to the secure verifier. Using the master check file previously generated, it locates the corresponding record in the check data produced by the mappers. If the hashes are not equal, verification fails and the sequence stops at this point. Otherwise, verification succeeds and we proceed to the next step.

**ShufflePhase** This job implements the shuffle phase. The output from the *RunMappers* program is partitioned, combined and sorted.

**SelectPartitionPairs** The program implements Algorithm 10.4 and runs on the secure verifier. It samples key-value pairs from the output generated by *ShufflePhase*. Again, a master check file is generated using a similar process as for *SelectRecords*.

**RunReducers** This job implements the reduce phase. Again, as each key-value pair is processed by an individual reducer, check data is written to a file. The data written is as in *RunMappers*.

**VerifyTasks** *VerifyTasks* also implements Algorithm 10.6. This time the input is the check data generated by the reducers and the master check file generated by *SelectPartitionPairs*. 
10.9 Experimental Results

To evaluate our algorithms, we performed a series of tests on a small Hadoop cluster consisting of one head node and three worker nodes. Each worker node was a Linux VM with one vCPU and 4GB of RAM. The head node is a Linux VM with one vCPU and 8GB RAM. Each test ran a variant of the canonical word count MR program where the probability that mappers and reducer would cheat is determined by the test configuration. If a mapper or reducer cheats, it then has a probability that it would alter its output with a probability also given by the test configuration. Similarly, each test configuration has a percentage of splits/partitions and records/pairs sampled by the verifier. Each test configuration was repeated 10 times. For each test configuration, the detection rates for the verifier and average timings for each stage are given in Tables 10.2 and 10.3.

We observe that detection rates most critically depend on the number of splits/partitions sampled. The more we sample the greater the detection rate. However, as we had a small number of nodes in our test environment, this undoubtedly made the number sampled more critical. The percentage of records/pairs the verifier samples seems to have an optimal value between 5 and 10%. This confirms our design goal that the verifier is capable of detecting cheating by sampling a fraction of the total data produced. Compare this with the methods of [39, 41, 235, 282, 337, 352] which use majority vote mechanisms which require considerable replication of work.

In addition, we ran the unaltered canonical word count program. This had an average running time of 13.15s. We also ran the a test configuration where no cheating occurs and no data is sampled by the verifier. This had an average total running time of 57.6s. This is 4.38 times slower. We can see from Tables 10.2 and 10.3 that this time further increases as we increase the amount of data sampled. This performance penalty is far worse than that of similar result verification methods discussed in section 3.3.2, which, where stated, are, at worst, a penalty of 78%. We attribute this to our implementation rather than the algorithms themselves. By implementing each algorithm as a separate MR job or program we have a large overhead as a result of Hadoop. In addition, our sequential verification/execution process negates any performance gains from streaming data.
from input and intermediate data. Furthermore, as a result of the strict MapReduce execution model, some of our jobs duplicate work already performed. The **RunMappers** job still has to parse the key-value pairs produced by **ReadRecords**. Similarly, **RunReducers** executes an identity mapper phase and then repeats the shuffle phase. These overheads could be eliminated if our verification algorithm is integrated into Hadoop itself.

Table 10.2: Timings for each experimental configuration of map phase cheating and verification. **Mappers** denotes the percentage of map tasks that are cheating. **Tamper** denotes the percentage of values altered by a cheating mapper. **Splits** denotes the percentage of input splits sampled by the verifier. **% Records** denotes the percentage of records in an input split sampled by the verifier. **Detect** denotes the detection rate of the verifier. **Parse** is the time to format the raw input as key-value pairs, **Sample** is the time the verifier takes to sample records from the input, **Map** is the time to execute all map tasks, **Verify** is the time for the verifier to verify the mapper check data against its own check data.

<table>
<thead>
<tr>
<th>Mappers (%)</th>
<th>Tamper (%)</th>
<th>Splits (%)</th>
<th>Records (%)</th>
<th>Detect (%)</th>
<th>Parse (s)</th>
<th>Sample (ms)</th>
<th>Map (s)</th>
<th>Verify (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>25</td>
<td>25</td>
<td>1</td>
<td>0</td>
<td>7.07</td>
<td>91.23</td>
<td>11.23</td>
<td>78.98</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>25</td>
<td>5</td>
<td>10</td>
<td>6.9</td>
<td>100.91</td>
<td>11.18</td>
<td>86.3</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>25</td>
<td>20</td>
<td>10</td>
<td>7.01</td>
<td>114.61</td>
<td>11.53</td>
<td>81.47</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>6.89</td>
<td>98.37</td>
<td>10.6</td>
<td>85.68</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>5</td>
<td>20</td>
<td>10</td>
<td>6.98</td>
<td>119.64</td>
<td>11.53</td>
<td>82.68</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>10</td>
<td>30</td>
<td>10</td>
<td>6.72</td>
<td>133.39</td>
<td>10.8</td>
<td>84.85</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>20</td>
<td>60</td>
<td>10</td>
<td>6.91</td>
<td>161.78</td>
<td>11</td>
<td>82.65</td>
</tr>
<tr>
<td>25</td>
<td>75</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>7.21</td>
<td>110.03</td>
<td>11.08</td>
<td>89.97</td>
</tr>
<tr>
<td>25</td>
<td>75</td>
<td>5</td>
<td>30</td>
<td>10</td>
<td>7.11</td>
<td>128.35</td>
<td>12.41</td>
<td>86.2</td>
</tr>
<tr>
<td>25</td>
<td>75</td>
<td>10</td>
<td>60</td>
<td>10</td>
<td>6.81</td>
<td>143.75</td>
<td>11.2</td>
<td>81.61</td>
</tr>
<tr>
<td>25</td>
<td>75</td>
<td>20</td>
<td>60</td>
<td>10</td>
<td>7</td>
<td>164.8</td>
<td>10.57</td>
<td>84.42</td>
</tr>
<tr>
<td>25</td>
<td>100</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>7.02</td>
<td>124.1</td>
<td>12.1</td>
<td>83.6</td>
</tr>
<tr>
<td>25</td>
<td>100</td>
<td>5</td>
<td>50</td>
<td>10</td>
<td>6.88</td>
<td>148.01</td>
<td>11.36</td>
<td>83.04</td>
</tr>
<tr>
<td>25</td>
<td>100</td>
<td>10</td>
<td>20</td>
<td>10</td>
<td>6.69</td>
<td>165.54</td>
<td>11.64</td>
<td>85.44</td>
</tr>
<tr>
<td>25</td>
<td>100</td>
<td>20</td>
<td>50</td>
<td>10</td>
<td>6.91</td>
<td>192.27</td>
<td>10.39</td>
<td>86.58</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>5</td>
<td>10</td>
<td>10</td>
<td>7.04</td>
<td>84.71</td>
<td>11.36</td>
<td>83.07</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>5</td>
<td>10</td>
<td>10</td>
<td>6.93</td>
<td>99.78</td>
<td>12.16</td>
<td>82.14</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>6.89</td>
<td>116.37</td>
<td>10.72</td>
<td>88.68</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>20</td>
<td>40</td>
<td>10</td>
<td>7.16</td>
<td>138.89</td>
<td>11.05</td>
<td>83.42</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>7.12</td>
<td>101.72</td>
<td>11.77</td>
<td>82.22</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>5</td>
<td>10</td>
<td>10</td>
<td>7.07</td>
<td>124.04</td>
<td>11.58</td>
<td>82.34</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>6.95</td>
<td>139.89</td>
<td>11.47</td>
<td>81.39</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>20</td>
<td>20</td>
<td>10</td>
<td>6.82</td>
<td>157</td>
<td>10.88</td>
<td>86.23</td>
</tr>
<tr>
<td>25</td>
<td>75</td>
<td>1</td>
<td>20</td>
<td>10</td>
<td>6.73</td>
<td>98.39</td>
<td>10.62</td>
<td>91.89</td>
</tr>
<tr>
<td>25</td>
<td>75</td>
<td>5</td>
<td>60</td>
<td>10</td>
<td>6.85</td>
<td>131.46</td>
<td>11.65</td>
<td>79.74</td>
</tr>
<tr>
<td>25</td>
<td>75</td>
<td>10</td>
<td>50</td>
<td>10</td>
<td>7.11</td>
<td>152.01</td>
<td>11.13</td>
<td>85.59</td>
</tr>
<tr>
<td>25</td>
<td>75</td>
<td>20</td>
<td>40</td>
<td>10</td>
<td>6.84</td>
<td>178.56</td>
<td>11.35</td>
<td>83.77</td>
</tr>
<tr>
<td>25</td>
<td>100</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>7.04</td>
<td>104.65</td>
<td>12.35</td>
<td>78.79</td>
</tr>
</tbody>
</table>
### 10.9. EXPERIMENTAL RESULTS

<table>
<thead>
<tr>
<th>25</th>
<th>50</th>
<th>100</th>
<th>5</th>
<th>20</th>
<th>6.58</th>
<th>139.95</th>
<th>10.93</th>
<th>81.48</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>50</td>
<td>100</td>
<td>10</td>
<td>70</td>
<td>7.23</td>
<td>168.78</td>
<td>11.79</td>
<td>86.17</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>100</td>
<td>20</td>
<td>60</td>
<td>7.12</td>
<td>190.49</td>
<td>11.81</td>
<td>83.23</td>
</tr>
<tr>
<td>25</td>
<td>90</td>
<td>25</td>
<td>1</td>
<td>0</td>
<td>7.08</td>
<td>94.19</td>
<td>11.58</td>
<td>83.46</td>
</tr>
<tr>
<td>25</td>
<td>90</td>
<td>25</td>
<td>5</td>
<td>0</td>
<td>6.82</td>
<td>110.98</td>
<td>10.93</td>
<td>82.01</td>
</tr>
<tr>
<td>25</td>
<td>90</td>
<td>25</td>
<td>10</td>
<td>20</td>
<td>7.01</td>
<td>111.32</td>
<td>12.43</td>
<td>82.16</td>
</tr>
<tr>
<td>25</td>
<td>90</td>
<td>25</td>
<td>20</td>
<td>30</td>
<td>6.67</td>
<td>134.97</td>
<td>11.45</td>
<td>81.32</td>
</tr>
<tr>
<td>25</td>
<td>90</td>
<td>50</td>
<td>1</td>
<td>0</td>
<td>6.97</td>
<td>99.82</td>
<td>11.3</td>
<td>78.48</td>
</tr>
<tr>
<td>25</td>
<td>90</td>
<td>50</td>
<td>5</td>
<td>0</td>
<td>6.78</td>
<td>112.07</td>
<td>10.9</td>
<td>81.72</td>
</tr>
<tr>
<td>25</td>
<td>90</td>
<td>50</td>
<td>10</td>
<td>0</td>
<td>6.91</td>
<td>137.35</td>
<td>11.24</td>
<td>81.15</td>
</tr>
<tr>
<td>25</td>
<td>90</td>
<td>50</td>
<td>10</td>
<td>40</td>
<td>6.91</td>
<td>137.35</td>
<td>11.24</td>
<td>81.15</td>
</tr>
<tr>
<td>25</td>
<td>90</td>
<td>75</td>
<td>1</td>
<td>10</td>
<td>7.06</td>
<td>104.85</td>
<td>11.48</td>
<td>80.35</td>
</tr>
<tr>
<td>25</td>
<td>90</td>
<td>75</td>
<td>5</td>
<td>50</td>
<td>6.88</td>
<td>127.07</td>
<td>11.32</td>
<td>84.74</td>
</tr>
<tr>
<td>25</td>
<td>90</td>
<td>75</td>
<td>10</td>
<td>70</td>
<td>6.99</td>
<td>158.66</td>
<td>11.32</td>
<td>81.47</td>
</tr>
<tr>
<td>25</td>
<td>90</td>
<td>75</td>
<td>20</td>
<td>60</td>
<td>7.10</td>
<td>170.51</td>
<td>11</td>
<td>84.65</td>
</tr>
<tr>
<td>25</td>
<td>90</td>
<td>100</td>
<td>1</td>
<td>10</td>
<td>6.8</td>
<td>107.29</td>
<td>11.76</td>
<td>88.11</td>
</tr>
<tr>
<td>25</td>
<td>90</td>
<td>100</td>
<td>5</td>
<td>60</td>
<td>6.95</td>
<td>141.25</td>
<td>11.52</td>
<td>81.72</td>
</tr>
<tr>
<td>25</td>
<td>90</td>
<td>100</td>
<td>10</td>
<td>80</td>
<td>6.81</td>
<td>157.77</td>
<td>11.77</td>
<td>80.67</td>
</tr>
<tr>
<td>25</td>
<td>90</td>
<td>100</td>
<td>20</td>
<td>60</td>
<td>6.86</td>
<td>189.3</td>
<td>11.47</td>
<td>81.69</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>25</td>
<td>1</td>
<td>0</td>
<td>6.83</td>
<td>85.28</td>
<td>10.56</td>
<td>79.99</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>25</td>
<td>5</td>
<td>10</td>
<td>6.78</td>
<td>104.42</td>
<td>11.31</td>
<td>82.27</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>25</td>
<td>10</td>
<td>0</td>
<td>6.98</td>
<td>113.12</td>
<td>11.5</td>
<td>81.73</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>25</td>
<td>20</td>
<td>40</td>
<td>7.06</td>
<td>122.36</td>
<td>11.44</td>
<td>87.41</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>50</td>
<td>1</td>
<td>10</td>
<td>6.81</td>
<td>103.17</td>
<td>11.05</td>
<td>80.74</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>50</td>
<td>5</td>
<td>20</td>
<td>6.82</td>
<td>125.55</td>
<td>10.86</td>
<td>81.71</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>50</td>
<td>10</td>
<td>60</td>
<td>6.9</td>
<td>141.89</td>
<td>11.31</td>
<td>81.64</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>50</td>
<td>20</td>
<td>60</td>
<td>7.16</td>
<td>149.93</td>
<td>11.65</td>
<td>87.69</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>75</td>
<td>1</td>
<td>10</td>
<td>7</td>
<td>102.5</td>
<td>11.27</td>
<td>84.55</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>75</td>
<td>5</td>
<td>40</td>
<td>6.78</td>
<td>139.43</td>
<td>11.92</td>
<td>81.41</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>75</td>
<td>10</td>
<td>80</td>
<td>6.75</td>
<td>161.58</td>
<td>11.3</td>
<td>81.8</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>75</td>
<td>20</td>
<td>90</td>
<td>7.15</td>
<td>177.08</td>
<td>11.03</td>
<td>80.84</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>100</td>
<td>1</td>
<td>20</td>
<td>6.98</td>
<td>117.02</td>
<td>11.61</td>
<td>81.73</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>100</td>
<td>5</td>
<td>80</td>
<td>6.74</td>
<td>145.85</td>
<td>11.46</td>
<td>81</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>100</td>
<td>10</td>
<td>70</td>
<td>6.84</td>
<td>172.26</td>
<td>10.48</td>
<td>84.84</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>100</td>
<td>20</td>
<td>70</td>
<td>6.88</td>
<td>184.99</td>
<td>11.49</td>
<td>83.23</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>25</td>
<td>1</td>
<td>0</td>
<td>6.96</td>
<td>91.59</td>
<td>11.53</td>
<td>79.22</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>25</td>
<td>5</td>
<td>10</td>
<td>6.85</td>
<td>98.4</td>
<td>10.57</td>
<td>83.29</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>25</td>
<td>10</td>
<td>60</td>
<td>6.88</td>
<td>116.12</td>
<td>10.96</td>
<td>80.46</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>25</td>
<td>20</td>
<td>30</td>
<td>6.68</td>
<td>128.47</td>
<td>11.35</td>
<td>86.54</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>50</td>
<td>1</td>
<td>20</td>
<td>6.79</td>
<td>100.39</td>
<td>11.49</td>
<td>81.1</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>50</td>
<td>5</td>
<td>40</td>
<td>7.09</td>
<td>110.89</td>
<td>11.29</td>
<td>88.31</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>50</td>
<td>10</td>
<td>50</td>
<td>6.91</td>
<td>117.41</td>
<td>11.83</td>
<td>82.24</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>50</td>
<td>20</td>
<td>90</td>
<td>6.81</td>
<td>149.59</td>
<td>12.69</td>
<td>87.23</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>75</td>
<td>1</td>
<td>10</td>
<td>6.94</td>
<td>102.67</td>
<td>10.99</td>
<td>78.81</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>75</td>
<td>5</td>
<td>50</td>
<td>7.14</td>
<td>121.66</td>
<td>10.2</td>
<td>82.96</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>75</td>
<td>10</td>
<td>60</td>
<td>6.92</td>
<td>139.68</td>
<td>11.81</td>
<td>83.27</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>75</td>
<td>20</td>
<td>80</td>
<td>7.26</td>
<td>165.36</td>
<td>10.99</td>
<td>85.67</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>100</td>
<td>1</td>
<td>10</td>
<td>6.76</td>
<td>121.1</td>
<td>11.78</td>
<td>79.3</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>100</td>
<td>5</td>
<td>80</td>
<td>6.76</td>
<td>137.12</td>
<td>11.58</td>
<td>82.26</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>100</td>
<td>10</td>
<td>50</td>
<td>6.99</td>
<td>162.91</td>
<td>11.14</td>
<td>81.39</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>100</td>
<td>20</td>
<td>90</td>
<td>6.76</td>
<td>184.7</td>
<td>10.64</td>
<td>81.3</td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>100</td>
<td>20</td>
<td>90</td>
<td>6.76</td>
<td>120.73</td>
<td>10.54</td>
<td>81.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>50</td>
<td>90</td>
<td>25</td>
<td>20</td>
<td>50</td>
<td>6.95</td>
<td>144.04</td>
<td>11.18</td>
<td>82.15</td>
</tr>
<tr>
<td>50</td>
<td>90</td>
<td>50</td>
<td>1</td>
<td>30</td>
<td>6.91</td>
<td>109.04</td>
<td>11.17</td>
<td>80.73</td>
</tr>
<tr>
<td>50</td>
<td>90</td>
<td>50</td>
<td>5</td>
<td>70</td>
<td>6.81</td>
<td>114.27</td>
<td>11.41</td>
<td>82.58</td>
</tr>
<tr>
<td>50</td>
<td>90</td>
<td>50</td>
<td>10</td>
<td>50</td>
<td>6.65</td>
<td>125.82</td>
<td>11.08</td>
<td>81.19</td>
</tr>
<tr>
<td>50</td>
<td>90</td>
<td>50</td>
<td>20</td>
<td>50</td>
<td>6.87</td>
<td>145.7</td>
<td>11.36</td>
<td>82.44</td>
</tr>
<tr>
<td>50</td>
<td>90</td>
<td>75</td>
<td>1</td>
<td>30</td>
<td>7.14</td>
<td>111.88</td>
<td>10.95</td>
<td>78.14</td>
</tr>
<tr>
<td>50</td>
<td>90</td>
<td>75</td>
<td>5</td>
<td>60</td>
<td>6.84</td>
<td>158.43</td>
<td>10.96</td>
<td>81.19</td>
</tr>
<tr>
<td>50</td>
<td>90</td>
<td>75</td>
<td>10</td>
<td>90</td>
<td>7.26</td>
<td>153.21</td>
<td>10.91</td>
<td>79.33</td>
</tr>
<tr>
<td>50</td>
<td>90</td>
<td>75</td>
<td>20</td>
<td>50</td>
<td>6.95</td>
<td>169.01</td>
<td>11.14</td>
<td>84</td>
</tr>
<tr>
<td>50</td>
<td>90</td>
<td>100</td>
<td>1</td>
<td>60</td>
<td>6.73</td>
<td>110.8</td>
<td>11.36</td>
<td>80.57</td>
</tr>
<tr>
<td>50</td>
<td>90</td>
<td>100</td>
<td>5</td>
<td>60</td>
<td>7.09</td>
<td>147.35</td>
<td>11.72</td>
<td>82.1</td>
</tr>
<tr>
<td>50</td>
<td>90</td>
<td>100</td>
<td>10</td>
<td>100</td>
<td>6.89</td>
<td>148.82</td>
<td>10.68</td>
<td>86.02</td>
</tr>
<tr>
<td>50</td>
<td>90</td>
<td>100</td>
<td>20</td>
<td>100</td>
<td>7.19</td>
<td>175.72</td>
<td>11.29</td>
<td>83.78</td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>25</td>
<td>1</td>
<td>0</td>
<td>7.01</td>
<td>83.67</td>
<td>11.44</td>
<td>79.75</td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>25</td>
<td>5</td>
<td>20</td>
<td>7.2</td>
<td>102.69</td>
<td>10.96</td>
<td>79.15</td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>25</td>
<td>10</td>
<td>50</td>
<td>6.82</td>
<td>127.26</td>
<td>11.5</td>
<td>87.06</td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>25</td>
<td>20</td>
<td>70</td>
<td>6.94</td>
<td>116.2</td>
<td>10.94</td>
<td>81.77</td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>50</td>
<td>1</td>
<td>40</td>
<td>7.09</td>
<td>104.33</td>
<td>10.69</td>
<td>81.86</td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>50</td>
<td>5</td>
<td>60</td>
<td>6.92</td>
<td>135.79</td>
<td>11.24</td>
<td>84.31</td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>50</td>
<td>10</td>
<td>70</td>
<td>7.27</td>
<td>160.11</td>
<td>12.21</td>
<td>89.45</td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>50</td>
<td>20</td>
<td>90</td>
<td>6.91</td>
<td>153.17</td>
<td>11.11</td>
<td>81.24</td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>75</td>
<td>1</td>
<td>10</td>
<td>6.74</td>
<td>100.54</td>
<td>11.42</td>
<td>78.48</td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>75</td>
<td>5</td>
<td>50</td>
<td>6.97</td>
<td>134.11</td>
<td>11.92</td>
<td>82.88</td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>75</td>
<td>10</td>
<td>80</td>
<td>6.99</td>
<td>146.2</td>
<td>11.37</td>
<td>85.13</td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>75</td>
<td>20</td>
<td>100</td>
<td>7.2</td>
<td>168.23</td>
<td>10.48</td>
<td>83.55</td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>100</td>
<td>1</td>
<td>20</td>
<td>7.09</td>
<td>114.81</td>
<td>11.2</td>
<td>82.63</td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>100</td>
<td>5</td>
<td>70</td>
<td>7.04</td>
<td>128.4</td>
<td>11.33</td>
<td>85.58</td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>100</td>
<td>10</td>
<td>100</td>
<td>7.03</td>
<td>164.23</td>
<td>11.16</td>
<td>81.49</td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>100</td>
<td>20</td>
<td>100</td>
<td>6.97</td>
<td>193.5</td>
<td>10.42</td>
<td>82.23</td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>25</td>
<td>1</td>
<td>0</td>
<td>6.89</td>
<td>95.01</td>
<td>10.07</td>
<td>80.31</td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>25</td>
<td>5</td>
<td>40</td>
<td>6.67</td>
<td>107.12</td>
<td>11.59</td>
<td>79.66</td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>25</td>
<td>10</td>
<td>90</td>
<td>7.01</td>
<td>123.66</td>
<td>11.36</td>
<td>80.32</td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>25</td>
<td>20</td>
<td>80</td>
<td>6.81</td>
<td>135.16</td>
<td>11.75</td>
<td>80.32</td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>50</td>
<td>1</td>
<td>40</td>
<td>6.86</td>
<td>104.51</td>
<td>11.14</td>
<td>83.54</td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>50</td>
<td>5</td>
<td>60</td>
<td>7.01</td>
<td>127.81</td>
<td>10.75</td>
<td>83.37</td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>50</td>
<td>10</td>
<td>90</td>
<td>6.71</td>
<td>142.44</td>
<td>11.56</td>
<td>86.17</td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>50</td>
<td>20</td>
<td>100</td>
<td>6.96</td>
<td>148.56</td>
<td>11.55</td>
<td>82.91</td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>75</td>
<td>1</td>
<td>20</td>
<td>7.05</td>
<td>108.12</td>
<td>11.82</td>
<td>78.97</td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>75</td>
<td>5</td>
<td>90</td>
<td>7.08</td>
<td>132.32</td>
<td>11.09</td>
<td>84.46</td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>75</td>
<td>10</td>
<td>90</td>
<td>6.77</td>
<td>162.91</td>
<td>11.04</td>
<td>85.41</td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>75</td>
<td>20</td>
<td>100</td>
<td>6.87</td>
<td>162.3</td>
<td>12</td>
<td>85.29</td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>100</td>
<td>1</td>
<td>50</td>
<td>7.05</td>
<td>122.15</td>
<td>11.63</td>
<td>84.08</td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>100</td>
<td>5</td>
<td>80</td>
<td>6.71</td>
<td>148.73</td>
<td>10.47</td>
<td>80.92</td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>100</td>
<td>10</td>
<td>100</td>
<td>6.62</td>
<td>169.65</td>
<td>10.23</td>
<td>83.74</td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>100</td>
<td>20</td>
<td>100</td>
<td>7.08</td>
<td>179.51</td>
<td>12.05</td>
<td>83.32</td>
</tr>
<tr>
<td>100</td>
<td>90</td>
<td>25</td>
<td>1</td>
<td>20</td>
<td>7.14</td>
<td>87.18</td>
<td>11.6</td>
<td>77.48</td>
</tr>
<tr>
<td>100</td>
<td>90</td>
<td>25</td>
<td>5</td>
<td>80</td>
<td>7.14</td>
<td>105.73</td>
<td>10.77</td>
<td>79.57</td>
</tr>
<tr>
<td>100</td>
<td>90</td>
<td>25</td>
<td>10</td>
<td>100</td>
<td>7.05</td>
<td>110.04</td>
<td>11.01</td>
<td>83.46</td>
</tr>
<tr>
<td>100</td>
<td>90</td>
<td>25</td>
<td>20</td>
<td>100</td>
<td>7.04</td>
<td>130.07</td>
<td>11.24</td>
<td>82.01</td>
</tr>
<tr>
<td>100</td>
<td>90</td>
<td>50</td>
<td>1</td>
<td>40</td>
<td>6.71</td>
<td>99.79</td>
<td>11.07</td>
<td>79.77</td>
</tr>
<tr>
<td>100</td>
<td>90</td>
<td>50</td>
<td>5</td>
<td>100</td>
<td>6.89</td>
<td>117.34</td>
<td>11.73</td>
<td>82.82</td>
</tr>
<tr>
<td>100</td>
<td>90</td>
<td>50</td>
<td>10</td>
<td>100</td>
<td>6.86</td>
<td>125.21</td>
<td>11.55</td>
<td>81.23</td>
</tr>
<tr>
<td>100</td>
<td>90</td>
<td>50</td>
<td>20</td>
<td>100</td>
<td>7.09</td>
<td>161.57</td>
<td>10.63</td>
<td>80.52</td>
</tr>
<tr>
<td>100</td>
<td>90</td>
<td>75</td>
<td>1</td>
<td>50</td>
<td>7.26</td>
<td>118.93</td>
<td>11.53</td>
<td>80.16</td>
</tr>
</tbody>
</table>
### 10.9. EXPERIMENTAL RESULTS

Table 10.3: Timings for each experimental configuration of reduce phase cheating and verification. Reducers denotes the percentage of reduce tasks that are cheating. Tamper denotes the percentage of values altered by a cheating reducer. Parts denotes the percentage of partitions sampled by the verifier. Pairs denotes the percentage of key-value pairs in a partition sampled by the verifier. Detect denotes the detection rate of the verifier. Shuffle is the time to perform the shuffle phase, Sample is the time the verifier takes to sample pairs from the partitions, Reduce is the time to execute all reduce tasks, Verify is the time for the verifier to verify the reducer check data against its own check data.

<table>
<thead>
<tr>
<th>Reducers (%)</th>
<th>Tamper (%)</th>
<th>Parts (%)</th>
<th>Pairs (%)</th>
<th>Detect (%)</th>
<th>Shuffle (s)</th>
<th>Sample (ms)</th>
<th>Reduce (s)</th>
<th>Verify (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>25</td>
<td>25</td>
<td>1</td>
<td>20</td>
<td>18.39</td>
<td>73.34</td>
<td>16.41</td>
<td>93.24</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>25</td>
<td>5</td>
<td>40</td>
<td>17.21</td>
<td>107.39</td>
<td>16.25</td>
<td>139.88</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>25</td>
<td>10</td>
<td>10</td>
<td>17.56</td>
<td>124.54</td>
<td>17.33</td>
<td>199.7</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>25</td>
<td>20</td>
<td>20</td>
<td>18.93</td>
<td>139.76</td>
<td>17.5</td>
<td>217.08</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>75</td>
<td>1</td>
<td>50</td>
<td>17.94</td>
<td>114.55</td>
<td>16.19</td>
<td>117.42</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>75</td>
<td>5</td>
<td>30</td>
<td>17.98</td>
<td>142.81</td>
<td>16.89</td>
<td>187.84</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>75</td>
<td>10</td>
<td>50</td>
<td>19.31</td>
<td>159.33</td>
<td>16.43</td>
<td>236.93</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>75</td>
<td>20</td>
<td>80</td>
<td>19.35</td>
<td>179.12</td>
<td>16.68</td>
<td>249.82</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>100</td>
<td>1</td>
<td>100</td>
<td>17.23</td>
<td>121.65</td>
<td>16.91</td>
<td>114.07</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>100</td>
<td>5</td>
<td>80</td>
<td>16.95</td>
<td>153.69</td>
<td>16.95</td>
<td>163.9</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>100</td>
<td>10</td>
<td>30</td>
<td>17.13</td>
<td>175.88</td>
<td>16.75</td>
<td>249.82</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>100</td>
<td>20</td>
<td>60</td>
<td>17.82</td>
<td>184.11</td>
<td>17.32</td>
<td>279.04</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>25</td>
<td>1</td>
<td>10</td>
<td>19.85</td>
<td>94.99</td>
<td>16.7</td>
<td>106.39</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>25</td>
<td>5</td>
<td>20</td>
<td>18</td>
<td>113</td>
<td>16.98</td>
<td>147.84</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>25</td>
<td>10</td>
<td>30</td>
<td>18.11</td>
<td>124.95</td>
<td>17.03</td>
<td>171.69</td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>25</td>
<td>20</td>
<td>30</td>
<td>19.84</td>
<td>137.08</td>
<td>16.47</td>
<td>199.82</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>50</td>
<td>1</td>
<td>50</td>
<td>17.97</td>
<td>106.69</td>
<td>17.45</td>
<td>104.96</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>50</td>
<td>5</td>
<td>20</td>
<td>18.51</td>
<td>134.4</td>
<td>17.42</td>
<td>176.37</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>50</td>
<td>10</td>
<td>30</td>
<td>18.58</td>
<td>146.18</td>
<td>16.17</td>
<td>213.98</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>50</td>
<td>20</td>
<td>30</td>
<td>17.68</td>
<td>166.17</td>
<td>16.58</td>
<td>273.12</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>75</td>
<td>1</td>
<td>70</td>
<td>18.51</td>
<td>116.2</td>
<td>16.83</td>
<td>116.43</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>75</td>
<td>5</td>
<td>50</td>
<td>18.52</td>
<td>140.39</td>
<td>17.44</td>
<td>189.59</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>75</td>
<td>10</td>
<td>90</td>
<td>18.22</td>
<td>154.13</td>
<td>17.64</td>
<td>159.57</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>75</td>
<td>20</td>
<td>80</td>
<td>17.38</td>
<td>183.94</td>
<td>17.21</td>
<td>197.84</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>100</td>
<td>1</td>
<td>70</td>
<td>17.75</td>
<td>120.43</td>
<td>16.59</td>
<td>124.95</td>
</tr>
<tr>
<td>Value</td>
<td>Value</td>
<td>Value</td>
<td>Value</td>
<td>Value</td>
<td>Value</td>
<td>Value</td>
<td>Value</td>
<td>Value</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>100</td>
<td>5</td>
<td>80</td>
<td>16.29</td>
<td>151.26</td>
<td>17.13</td>
<td>144.67</td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>10</td>
<td>70</td>
<td>17.97</td>
<td>170.1</td>
<td>17.7</td>
<td>219.38</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>20</td>
<td>70</td>
<td>18.17</td>
<td>186.31</td>
<td>16.59</td>
<td>283.62</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>25</td>
<td>1</td>
<td>20</td>
<td>19.32</td>
<td>104.81</td>
<td>16.76</td>
<td>104.31</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>25</td>
<td>5</td>
<td>20</td>
<td>17.5</td>
<td>115.65</td>
<td>17.83</td>
<td>150.18</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>25</td>
<td>10</td>
<td>30</td>
<td>19.84</td>
<td>128.82</td>
<td>17.32</td>
<td>174.54</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>25</td>
<td>20</td>
<td>50</td>
<td>17.25</td>
<td>140.6</td>
<td>16.91</td>
<td>172.58</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>50</td>
<td>1</td>
<td>40</td>
<td>19.09</td>
<td>105.61</td>
<td>16.76</td>
<td>119.95</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>50</td>
<td>5</td>
<td>40</td>
<td>17.3</td>
<td>130.55</td>
<td>16.66</td>
<td>166.73</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>50</td>
<td>10</td>
<td>60</td>
<td>17.43</td>
<td>147.44</td>
<td>17.36</td>
<td>188.59</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>50</td>
<td>20</td>
<td>20</td>
<td>19.17</td>
<td>163.08</td>
<td>16.74</td>
<td>265.39</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>75</td>
<td>1</td>
<td>30</td>
<td>17.52</td>
<td>113.48</td>
<td>16.99</td>
<td>134.82</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>75</td>
<td>5</td>
<td>40</td>
<td>17.65</td>
<td>141.61</td>
<td>16.76</td>
<td>193.1</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>75</td>
<td>10</td>
<td>50</td>
<td>17.42</td>
<td>158.97</td>
<td>16.8</td>
<td>256.71</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>75</td>
<td>20</td>
<td>50</td>
<td>18.73</td>
<td>173.15</td>
<td>17.31</td>
<td>259.31</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>100</td>
<td>1</td>
<td>40</td>
<td>17.74</td>
<td>121.89</td>
<td>16.64</td>
<td>136.33</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>100</td>
<td>5</td>
<td>70</td>
<td>18.01</td>
<td>157.16</td>
<td>16.67</td>
<td>185.83</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>100</td>
<td>10</td>
<td>40</td>
<td>18.6</td>
<td>169.96</td>
<td>16.59</td>
<td>269.65</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>100</td>
<td>20</td>
<td>60</td>
<td>17.7</td>
<td>189.41</td>
<td>17.36</td>
<td>286.89</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>1</td>
<td>80</td>
<td>17.65</td>
<td>99.91</td>
<td>16.89</td>
<td>90.11</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>5</td>
<td>70</td>
<td>17.26</td>
<td>121.2</td>
<td>16.59</td>
<td>106.42</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>10</td>
<td>30</td>
<td>17.65</td>
<td>122.76</td>
<td>16.67</td>
<td>164.48</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>25</td>
<td>20</td>
<td>30</td>
<td>17.26</td>
<td>137.57</td>
<td>16.42</td>
<td>204.96</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>1</td>
<td>70</td>
<td>18.3</td>
<td>105.67</td>
<td>17.32</td>
<td>116.33</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>5</td>
<td>90</td>
<td>18.1</td>
<td>130.4</td>
<td>16.11</td>
<td>109.06</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>10</td>
<td>70</td>
<td>16.78</td>
<td>153.11</td>
<td>16.29</td>
<td>141.85</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>50</td>
<td>20</td>
<td>50</td>
<td>20.37</td>
<td>159.52</td>
<td>16.83</td>
<td>200.56</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>75</td>
<td>1</td>
<td>80</td>
<td>18.37</td>
<td>110.81</td>
<td>15.99</td>
<td>102.21</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>75</td>
<td>5</td>
<td>90</td>
<td>17</td>
<td>143.25</td>
<td>17.46</td>
<td>132.15</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>75</td>
<td>10</td>
<td>90</td>
<td>17.25</td>
<td>160.61</td>
<td>16.62</td>
<td>137.56</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>75</td>
<td>20</td>
<td>90</td>
<td>16.47</td>
<td>175.8</td>
<td>17.02</td>
<td>155.26</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>100</td>
<td>1</td>
<td>90</td>
<td>18.34</td>
<td>120.64</td>
<td>16.65</td>
<td>112.07</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>100</td>
<td>5</td>
<td>90</td>
<td>18.36</td>
<td>153.57</td>
<td>16.63</td>
<td>141.49</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>100</td>
<td>10</td>
<td>100</td>
<td>19.07</td>
<td>170.62</td>
<td>17.75</td>
<td>181.09</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>100</td>
<td>20</td>
<td>100</td>
<td>17.59</td>
<td>186.17</td>
<td>17.87</td>
<td>172.36</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>1</td>
<td>70</td>
<td>16.5</td>
<td>96.02</td>
<td>17.07</td>
<td>92.76</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>5</td>
<td>50</td>
<td>17.64</td>
<td>116.44</td>
<td>16.75</td>
<td>125.59</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>10</td>
<td>30</td>
<td>18.21</td>
<td>123.75</td>
<td>16.67</td>
<td>169.3</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>20</td>
<td>60</td>
<td>18.35</td>
<td>137.98</td>
<td>16.57</td>
<td>149.56</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>1</td>
<td>70</td>
<td>17.89</td>
<td>108.69</td>
<td>16.78</td>
<td>112.31</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>5</td>
<td>70</td>
<td>18.31</td>
<td>135.93</td>
<td>16.96</td>
<td>125.25</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>10</td>
<td>70</td>
<td>18.25</td>
<td>148.2</td>
<td>16.75</td>
<td>177.9</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>20</td>
<td>80</td>
<td>17.06</td>
<td>160.95</td>
<td>16.46</td>
<td>197.95</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>75</td>
<td>1</td>
<td>80</td>
<td>17.98</td>
<td>113.88</td>
<td>16.4</td>
<td>105.28</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>75</td>
<td>5</td>
<td>80</td>
<td>16.65</td>
<td>143.1</td>
<td>16.36</td>
<td>132.12</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>75</td>
<td>10</td>
<td>100</td>
<td>16.49</td>
<td>159.09</td>
<td>16.91</td>
<td>159.78</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>75</td>
<td>20</td>
<td>80</td>
<td>16.95</td>
<td>173.77</td>
<td>16.4</td>
<td>232.41</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>100</td>
<td>1</td>
<td>100</td>
<td>17.73</td>
<td>118.82</td>
<td>16.96</td>
<td>96.29</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>100</td>
<td>5</td>
<td>90</td>
<td>17.4</td>
<td>149.02</td>
<td>16.46</td>
<td>139.58</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>100</td>
<td>10</td>
<td>90</td>
<td>17.79</td>
<td>170.43</td>
<td>16.59</td>
<td>185.82</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>100</td>
<td>20</td>
<td>100</td>
<td>17.01</td>
<td>193.51</td>
<td>16.78</td>
<td>207.26</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>25</td>
<td>1</td>
<td>50</td>
<td>18.37</td>
<td>103.06</td>
<td>16.98</td>
<td>99.4</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>25</td>
<td>5</td>
<td>50</td>
<td>18.08</td>
<td>119.09</td>
<td>17.06</td>
<td>119.03</td>
<td></td>
</tr>
<tr>
<td>90</td>
<td>25</td>
<td>10</td>
<td>60</td>
<td>17.09</td>
<td>124.68</td>
<td>16.52</td>
<td>129.2</td>
<td></td>
</tr>
</tbody>
</table>
## 10.9. EXPERIMENTAL RESULTS

<table>
<thead>
<tr>
<th>50</th>
<th>90</th>
<th>25</th>
<th>20</th>
<th>50</th>
<th>16.76</th>
<th>142.71</th>
<th>16.81</th>
<th>167.12</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>90</td>
<td>50</td>
<td>1</td>
<td>70</td>
<td>17.63</td>
<td>110.66</td>
<td>18.61</td>
<td>102.39</td>
</tr>
<tr>
<td>50</td>
<td>90</td>
<td>50</td>
<td>5</td>
<td>80</td>
<td>17.83</td>
<td>139.67</td>
<td>18.77</td>
<td>136.94</td>
</tr>
<tr>
<td>50</td>
<td>90</td>
<td>50</td>
<td>10</td>
<td>70</td>
<td>17.16</td>
<td>147.7</td>
<td>17.16</td>
<td>145.74</td>
</tr>
<tr>
<td>50</td>
<td>90</td>
<td>50</td>
<td>20</td>
<td>80</td>
<td>18.58</td>
<td>161.55</td>
<td>17.38</td>
<td>154.08</td>
</tr>
<tr>
<td>50</td>
<td>90</td>
<td>75</td>
<td>1</td>
<td>90</td>
<td>19.66</td>
<td>131.64</td>
<td>18.77</td>
<td>181.73</td>
</tr>
<tr>
<td>50</td>
<td>90</td>
<td>75</td>
<td>5</td>
<td>100</td>
<td>21.24</td>
<td>185.38</td>
<td>19.61</td>
<td>191.52</td>
</tr>
<tr>
<td>50</td>
<td>90</td>
<td>75</td>
<td>10</td>
<td>100</td>
<td>20.04</td>
<td>204.98</td>
<td>20.21</td>
<td>193.14</td>
</tr>
<tr>
<td>50</td>
<td>90</td>
<td>75</td>
<td>20</td>
<td>90</td>
<td>20.85</td>
<td>208.07</td>
<td>19.37</td>
<td>240.22</td>
</tr>
<tr>
<td>50</td>
<td>100</td>
<td>1</td>
<td>1</td>
<td>100</td>
<td>19.53</td>
<td>136.88</td>
<td>17.7</td>
<td>116.58</td>
</tr>
<tr>
<td>50</td>
<td>100</td>
<td>5</td>
<td>90</td>
<td>20.35</td>
<td>188.3</td>
<td>17.91</td>
<td>164.53</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>100</td>
<td>10</td>
<td>90</td>
<td>18.63</td>
<td>201.23</td>
<td>18.47</td>
<td>171.66</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>100</td>
<td>20</td>
<td>100</td>
<td>21.04</td>
<td>213.07</td>
<td>19.18</td>
<td>206.05</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>1</td>
<td>100</td>
<td>22.16</td>
<td>145.54</td>
<td>20.96</td>
<td>121.43</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>5</td>
<td>100</td>
<td>22.04</td>
<td>148.38</td>
<td>18.17</td>
<td>102.73</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>10</td>
<td>100</td>
<td>19.4</td>
<td>157.59</td>
<td>18.38</td>
<td>100.72</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>20</td>
<td>100</td>
<td>18.65</td>
<td>157.27</td>
<td>18.44</td>
<td>97.05</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>1</td>
<td>100</td>
<td>18.93</td>
<td>126.79</td>
<td>17.41</td>
<td>97.37</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>5</td>
<td>100</td>
<td>17.96</td>
<td>139.87</td>
<td>18.34</td>
<td>93.53</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>10</td>
<td>100</td>
<td>17.69</td>
<td>146.62</td>
<td>17.21</td>
<td>89.66</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>20</td>
<td>100</td>
<td>17.46</td>
<td>165.1</td>
<td>16.84</td>
<td>89.52</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>75</td>
<td>1</td>
<td>100</td>
<td>16.26</td>
<td>117.94</td>
<td>17.71</td>
<td>109.53</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>75</td>
<td>5</td>
<td>100</td>
<td>18.65</td>
<td>146.91</td>
<td>17.27</td>
<td>85.25</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>75</td>
<td>10</td>
<td>100</td>
<td>16.77</td>
<td>157.99</td>
<td>16.62</td>
<td>92.07</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>75</td>
<td>20</td>
<td>100</td>
<td>17.08</td>
<td>182.13</td>
<td>16.79</td>
<td>84.95</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>1</td>
<td>100</td>
<td>17.67</td>
<td>124.97</td>
<td>17.06</td>
<td>89.64</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>5</td>
<td>100</td>
<td>17.93</td>
<td>154.55</td>
<td>16.75</td>
<td>85.01</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>10</td>
<td>100</td>
<td>18.33</td>
<td>177.81</td>
<td>16.6</td>
<td>90.97</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>20</td>
<td>100</td>
<td>18.58</td>
<td>189.54</td>
<td>16.69</td>
<td>87.31</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>1</td>
<td>100</td>
<td>19.41</td>
<td>102.31</td>
<td>17.52</td>
<td>84.2</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>5</td>
<td>100</td>
<td>18.54</td>
<td>116.02</td>
<td>16.77</td>
<td>84.1</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>10</td>
<td>100</td>
<td>17.94</td>
<td>128.41</td>
<td>18.04</td>
<td>83.7</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>20</td>
<td>100</td>
<td>18.66</td>
<td>142.27</td>
<td>17.22</td>
<td>84.47</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>1</td>
<td>100</td>
<td>18.73</td>
<td>112.11</td>
<td>17.11</td>
<td>86.25</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>5</td>
<td>100</td>
<td>18.87</td>
<td>134.52</td>
<td>16.75</td>
<td>84.93</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>10</td>
<td>100</td>
<td>17.9</td>
<td>158.6</td>
<td>16.94</td>
<td>85.17</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>20</td>
<td>100</td>
<td>18.9</td>
<td>163.41</td>
<td>18.29</td>
<td>83.82</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>75</td>
<td>1</td>
<td>100</td>
<td>17.35</td>
<td>121.03</td>
<td>16.7</td>
<td>86.01</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>75</td>
<td>5</td>
<td>100</td>
<td>16.9</td>
<td>151.91</td>
<td>16.51</td>
<td>84.89</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>75</td>
<td>10</td>
<td>100</td>
<td>17.29</td>
<td>163.75</td>
<td>16.6</td>
<td>88.16</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>75</td>
<td>20</td>
<td>100</td>
<td>17.58</td>
<td>180.41</td>
<td>16.77</td>
<td>84.64</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>1</td>
<td>100</td>
<td>16.91</td>
<td>124.55</td>
<td>17</td>
<td>85.39</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>5</td>
<td>100</td>
<td>16.91</td>
<td>150.1</td>
<td>17.15</td>
<td>86.04</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>10</td>
<td>100</td>
<td>18.63</td>
<td>170.01</td>
<td>16.84</td>
<td>87.47</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>20</td>
<td>100</td>
<td>17.75</td>
<td>189.96</td>
<td>16.85</td>
<td>84.87</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>1</td>
<td>80</td>
<td>18.2</td>
<td>98.64</td>
<td>16.71</td>
<td>97.06</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>5</td>
<td>50</td>
<td>18.07</td>
<td>115.79</td>
<td>16.75</td>
<td>122.87</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>10</td>
<td>60</td>
<td>18.29</td>
<td>127.02</td>
<td>17.14</td>
<td>137.1</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>25</td>
<td>20</td>
<td>80</td>
<td>20.37</td>
<td>145.69</td>
<td>17.42</td>
<td>116.12</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>1</td>
<td>100</td>
<td>17.97</td>
<td>114.65</td>
<td>17.16</td>
<td>89.22</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>5</td>
<td>90</td>
<td>18.1</td>
<td>129.89</td>
<td>16.86</td>
<td>125.2</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>10</td>
<td>80</td>
<td>18.8</td>
<td>150.55</td>
<td>16.86</td>
<td>139.7</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>20</td>
<td>100</td>
<td>16.98</td>
<td>167.02</td>
<td>16.97</td>
<td>122.64</td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>75</td>
<td>1</td>
<td>100</td>
<td>19.06</td>
<td>115.72</td>
<td>16.65</td>
<td>88.58</td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 10. RESULT VERIFICATION

10.10 Conclusion

This chapter has presented a conceptually simple probabilistic method to verify the results of mappers and reducers. As we state earlier in the chapter, such a system is important for CryptMR because if any input or intermediate datum is tampered with, the result of the computation will not decrypt to the correct value. While CryptMR prevents snooping on data it does not prevent tampering.

In our verification system, a verifier samples records/pairs from the input split/partition, applies the map/reduce function to the sampled data. Using the output it received, it checks this output against the output produced by the mapper or reducer to detect cheating.

The goal of this work was to produce a verification method which, with strong probability, attests whether cheating has occurred or not by sampling a small amount of data. Our experimental results demonstrate that, in this respect, we have achieved this goal.

However, it should be noted that our implementation is a “proof of concept” and is, currently, too slow for practical use. We can investigate integrating our verification algorithms into Hadoop so that the performance penalties described above might be eliminated. It should be noted, however, that our current implementation does guarantee that selection of verification data and verification can be performed in a secure environment. This might not be possible in an integrated solution. We also note that our implemented solution makes it possible to detect a cheating node. Therefore, our solution could be enhanced to support blacklisting of cheating nodes. This would mean that the computation could continue rather than being aborted as in the current implementation.

It should also be noted that while our solution is generic and can deal with any
MapReduce applications, we can further enhance it by adding application specific watermarked data as in [166] or [333].
Chapter 11

Conclusion

This thesis has detailed work undertaken toward a practical secure computation in the cloud system using MapReduce. In chapters 1 and 2, we introduced the problem, its motivation, and its background. In chapter 3, we present a review of the literature concerning appropriate cryptographic techniques and related work concerning MapReduce security. In chapter 4, we produce a generic model of MapReduce computation along with an extensive threat analysis of this model. In chapters 6, 7, 8, and 9, we present details of CryptMR, our secure computation in the cloud using MapReduce solution. Finally, in chapter 10, we present a result verification method for MapReduce.

In section 1.4, we detailed the novel contributions of this thesis. These were: an extensive threat analysis of the interaction between components of the MapReduce framework; several new encryption schemes designed for use in secure computation in the cloud; and a method to verify the results of a MapReduce computation by limited duplication of work.

The threat analysis (chapter 4) we conducted focused on communication and data flow between the entities involved in a MapReduce job execution. When this material was first published [122], this was the first such treatment available. While a later paper [109] adds to our work, particularly with regard to surveying the literature on MapReduce security, the threats they identify are the same as those identified in our work.

We have also provided several novel encryption methods that are appropriate to
large-scale cloud computing. We have presented two hierarchies of novel “some-
what” homomorphic encryption over the integers, of increasing security, HEk and
HEkN, in chapter 7. The HEk hierarchy encrypt integers to a k-vector, where
each element is a large integer modulo the product of two large primes. The
HEkN variant is similar, but adds an additional “noise” term to the vector ele-
ments. We discuss four variants in detail, HE1, HE1N, HE2, and HE2N, for which
we present extensive experimental results which are up to 1000 times faster than
those reported in the literature for related work. Furthermore, we have presen-
ted results for large-scale experiments performed in Microsoft’s Azure cloud that
demonstrate that our systems scale well to a large number of inputs. This is
of particular importance when considering that we intend for our cryptographic
systems to be used in big data analysis.

In chapter 8, we detail our novel randomised order-preserving encryption (OPE)
scheme which uses the general approximate common divisor problem [164] as its
basis. We believe that this is the first such scheme to be based on a computa-
tionally hard primitive. Again, we provide extensive experimental results for
our algorithms both in small- and large-scale cloud environment with extremely
favourable results in comparison with other OPE schemes.

In chapter 9 we present a novel symmetric searchable encryption scheme. This
scheme builds on the work of Song et al. [309] by addressing some of the imprac-
ticalities of their scheme.

In addition to the work on encryption schemes, we have also produced a novel
result verification method for MapReduce. While computing over encrypted data
guarantees that only limited information, such as the operations performed on the
data, are leaked, it does not guarantee that the computational results have not
been tampered with. The work in chapter 10 discusses a probabilistic method
for verifying the results of a MapReduce computation by sampling relatively
small numbers of results and checking them against pre-computed results. We
show that sampling between 5 and 10% of results yields a strong probability that
tampering is detected if it has occurred.

With the exception of the threat analysis, these contributions are core components
of our secure computation in the cloud solution, CryptMR. As argued in chapter

---

1This work was supported by a $20,000 Microsoft Azure for Research sponsorship. This
sponsorship may be used for further research.
we believe that computing over encrypted data is the only way to successfully negate the threat of the privileged insider, such as a cloud provider system administrator, with regard to outsourcing computation in the cloud. While the approach of fully homomorphic encryption would theoretically allow one to compute any function over encrypted data, the running times currently presented in the literature are far too slow for practical use (see section 3.2.1). Therefore, our approach was to devise a system for computation over encrypted data where the running times to compute functions typically used by many application are much closer to those for the same computation on unencrypted data. In order to achieve this, we adopted a similar approach to CryptDB [272], in that we have chosen to tie the method of encryption of data to the operation to be performed on that data. In this way, we believe that we have devised cryptosystems which allow us to perform computation over encrypted data but which have far less performance penalty than FHE in comparison with the same computation on unencrypted data. However, the encryption schemes we have devised only allow for a limited set of operations to be performed on the encrypted data and are mutually incompatible with each other. This a major limitation of our work as one cannot perform, say, arithmetic operations on encrypted data and then sort the encryptions, and expect correctly sorted output when decrypted. We should note that this limitation is also a problem for related work [272, 311, 313] which we have discussed in chapter 3. While this is not an issue for FHE schemes, they are magnitudes slower than our schemes and do not seem likely to become much faster in the near future. While we note that, in comparison to related work [272, 311, 313], our HEk encryption schemes have extended arithmetic operations to both addition and multiplication while outperforming the schemes used in those solutions (see 7.7), we see a trade-off existing between flexibility of computation and speed of computation. We have opted to focus on the latter. However, we should note that, with regard to MapReduce, this limitation could be overcome by several rounds of MapReduce where the output from one round is re-encrypted so as to be suitable for the next round, although such an approach would be considerably slower than computing on unencrypted data.

With regard to the timings of our algorithms, they currently lie in the range of several microseconds to several milliseconds depending on the algorithm used and the function computed. This is still some way off the several teraflops that modern CPUs and GPUs are capable of. Reducing those timings to several nanoseconds...
will be the direction of future work, as this would take us several magnitudes closer to the timings for computing on unencrypted data.

We should also note that the encryption schemes in CryptMR are entirely generic. They may be applied to any scenario where computing over encrypted data is required, rather than just for use in MapReduce. Therefore, they could be applied to other distributed or non-distributed computing platforms. Our Hadoop Writable library (see Appendix A) serves as a reference for how CryptMR can be integrated into a distributed computing platform. Furthermore, as a result of this generic nature, the encryption schemes used in CryptMR can be easily ported to other related work such as MrCrypt [313], Crypsis [311], or CryptDB [272].

We conclude each of chapters 6 to 10 with a discussion of further work that may be undertaken to enhance each component of CryptMR. Investigating these ideas will be the basis of future work on this project. However, there are improvements to CryptMR as a whole that can also be made. We use Java for our implementation, which is natural given its relationship to Hadoop. If we re-implemented our work it might lead to significant performance gains. We should also note that our CryptMR system provides the basic building blocks for building an application that computes over encrypted data. This is by design, we intended to support as many different applications as possible. However, we could provide implementations of suitable common algorithms to better facilitate third party application development.

Therefore, to conclude, we believe that our work demonstrates significant progress towards a practical secure computation in the cloud system using MapReduce.
Bibliography


[6] Mike Aizatsky, Chris Bunch, Brett Slatkin, Tom Kaitchuk, John Lucena, Josh Whelchel, Matt Faus, Kevin Sookocheff, Tony Liu, Mark Goldstein, Craig Silverstein, Damien Lebrun, Sean O’Keefe, Andrei Safronau, Chris Vaughn, Xiaolong Cheng, Tom Yedwab, Dmitry Sadovnychyi and


BIBLIOGRAPHY


[315] Iain Thomson. Microsoft researchers smash homomorphic encryption speed barrier. 9th Feb. 2016. URL: https://www.theregister.co.uk/2016/02/09/researchers_break_homomorphic_encryption/.


Jiaqi Zhao, Lizhe Wang, Jie Tao, Jinjun Chen, Weiye Sun, Rajiv Ranjan, Joanna Kołodziej, Achim Streit and Dimitrios Georgakopoulos. ‘A security framework in G-Hadoop for big data computing across distributed Cloud


Appendix A

Implementation of CryptMR

A.1 Introduction

In this appendix, we present details of the implementation of CryptMR.

CryptMR is divided into two pieces of software: a library of Java classes which implement the algorithms described in chapters 7 to 9 and a library which implements the ciphertexts generated by those algorithms as Hadoop Writable classes.

A.2 CryptMR Encryption Library

Each algorithm presented in earlier chapters is implemented as two Java classes: an Encrypter class which implements the encryption algorithm and a Decrypter class which implements the decryption algorithm. Key and parameter generation for each algorithm is implemented in the constructor for the Encrypter class. The secrets and public parameters generated are written as JSON strings to files. The secrets file is stored on the secure client and read by the Decrypter class to construct the decryption algorithm. The public parameters file is used by the pertinent Hadoop Writable class.

In all the algorithms detailed, pseudorandom numbers are generated using the
A.2. CRYPTMR ENCRYPTION LIBRARY

Figure A.1: UML diagram for the Java pseudorandom number generator (PRNG) class

ISAAC cipher. We have used a subclass of the rosettacode.org Java implementation which implements auxiliary functions for generating pseudorandom Java BigIntegers. The UML diagram for these classes is shown in Figure A.1.
A.2.1 Implementations of RND and DET Schemes

The RND and DET schemes are implemented as the Java classes `DETEncrypter`, `DETDencrypter`, `RNDEncrypter` and `RNDDecrypter`. Each uses AES-256 in CBC mode using PKCS#7 padding. They are implemented using the BouncyCastle cryptographic library. DETEncrypter uses a constant key and initialization vector (IV). RNDEncrypter generates a new key and random IV for each encryption. For RND, the AES key and IV are further encrypted using RSA public key encryption. The encrypted AES key, IV, and the ciphertext generated by AES are written to a JSON string. A UML diagram of the Java class hierarchy for the RND and DET implementations is shown in Figure A.2.

A.2.2 Implementation of the HOM Scheme

The HOM scheme is given by a family of algorithms, $HE_k$ and $HE_kN$, which are detailed in chapter 7. We only implement algorithms $HE_1$, $HE_1N$, $HE_2$ and $HE_2N$. Again, each algorithm is implemented as two classes: an Encrypter and Decrypter. We employed the JScience library for arbitrary precision integer support using their `LargeInteger` and `ModuloInteger` classes. For the vector based schemes we also use JScience’s `DenseVector` and `DenseMatrix` classes. UML diagrams for the class hierarchy are given in Figures A.3, A.4, A.5, A.6 and A.7.

A.2.3 Implementation of the OPE Scheme

The OPE scheme is again implemented as two Java classes. We use Java `BigInteger` for arbitrary precision integer support. A UML diagram is shown in Figure A.8.

A.2.4 Implementation of the SEARCH Scheme

The SEARCH scheme is implemented as two Java classes. The $SE$ scheme is implemented using the same algorithms as the RND scheme above. The $MAC$ scheme is implemented using the CBC-MAC algorithm built on AES. The
pseudorandom function $f$ is constructed using AES in the same fashion as the DET scheme above. The UML diagram for the Java classes is shown in Figure A.9.

A.3 CryptMR Hadoop Writable Library

This library provides datatypes for the ciphertexts produced by the schemes detailed above. All these classes implement the Hadoop Writable interface. The Writable specifies the interface that Hadoop’s lightweight serialization and deserialization processes will use, allowing Hadoop to efficiently transfer object data between MapReduce tasks and over-the-wire to other cluster nodes. Any datatype used in Hadoop must implement this interface.

In the case of RNDWritable, this class is simply a convenient wrapper around the ciphertext. OPEWritable simply implements the Writable interface for arbitrary precision integers. DETWritable is a subclass of RNDWritable but also implements the equality test. For (HE1Writable, HE2Writable and SSEWritable), the Writable type also implements the publicly visible homomorphic functions such as addition and multiplication for the HE$k$ and HE$k$N schemes. Where a Writable requires access to the public parameters of the cipher, for example, HE2 has two public parameters: the arithmetic modulus and the re-encryption matrix $R$; these parameters are read in from the JSON string in the parameters file generated by the encryption class. Figure A.10 shows the Java Writable classes.
Figure A.2: UML Diagram for Java class implementations of RND and DET.
Figure A.3: UML diagram for the parent Java classes of the HOM implementation
Figure A.4: UML diagram for Java class implementations of HE1

Figure A.5: UML diagram for Java class implementations of HE1N
Figure A.6: UML diagram for Java class implementations of HE2

Figure A.7: UML diagram for Java class implementations of HE2N
Figure A.8: UML diagram for Java class implementations of OPE
Figure A.9: UML diagram for Java class implementations of SEARCH
Figure A.10: UML diagram for the Java Hadoop Writable classes
Appendix B

Proofs

B.1 Chapter 7

Theorem 1. An attack against HE1 is successful in polynomial time if and only if we can factorise a distinct semi-prime in polynomial time.

Proof. Suppose that we have an unknown plaintext $m$, encrypted as $c = m + rp \mod pq$, where $r \leftarrow [1, q)$.

If we can factor $pq$ in polynomial time, we can determine $p$ and $q$ in polynomial time, since we know $p < q$. Therefore, we can determine $m = c \mod p$.

If we can determine $m$ given $c$ for arbitrary $m$, then we can determine $rp = c - m$.

We are given $q$, and we know $0 < r < q$, so $\gcd(rp, q)$ must be $p$, and we can compute $p$ in polynomial time. Now, given $p$, we can determine $q$ as $qp/p$. Hence, we can factorise $pq$ in polynomial time.

Lemma 2. If the inputs $m$ have entropy $\rho$ then, for any two independent inputs $m_1, m_2$, $\Pr(m_1 = m_2) \leq 2^{-\rho}$.

Proof. $\Pr(m_1 = m_2) = \sum_{i=0}^{M-1} \xi_i^2 = 2^{-H_2} \leq 2^{-\rho}$, since $H_2 \geq H_\infty = \rho$.

Theorem 3. For any encryption $c$, $c \mod \kappa$ is polynomial time indistinguishable from the uniform distribution on $[0, \kappa)$. Thus HE1N satisfies IND-CPA, under the assumption that SPACDP is not polynomial time solvable.
Appendix B. Proofs

Proof.

\[ c = m + sk + rp = m + rp \mod \kappa, \]

where \( r \xleftarrow{\$} [1, q) \). Thus, for \( i \in [0, \kappa) \),

\[
\Pr(c \mod \kappa = i) = \Pr(m + rp = i \mod \kappa) \\
= \Pr(r = p^{-1}(i - m) \mod \kappa) \\
\in \{\lfloor q/\kappa \rfloor 1/q, \lfloor q/\kappa \rfloor 1/q\} \\
\in [1/\kappa - 1/q, 1/\kappa + 1/q],
\]

where the inverse \( p^{-1} \) of \( p \mod \kappa \) exists since \( p \) is a prime. Hence the total variation distance from the uniform distribution is

\[
\frac{1}{2} \sum_{i=0}^{\kappa-1} \left| \Pr(c \mod \kappa = i) - 1/\kappa \right| < \kappa/q.
\]

This is exponentially small in the security parameter \( \lambda \) of the system, so the distribution of \( c \mod \kappa \) cannot be distinguished in polynomial time from the uniform distribution. Note further that \( c_1 \mod \kappa, c_2 \mod \kappa \) are independent for any two ciphertexts \( c_i = m_i + s_i \kappa + r_ip \) \( (i = 1, 2) \), since \( r_1, r_2 \) are independent.

To show IND-CPA, suppose now that known plaintexts \( \mu_1, \ldots, \mu_n \) are encrypted by an oracle for HE1N, giving ciphertexts \( c_1, \ldots, c_n \). Then, for \( r_i \xleftarrow{\$} [0, q) \), \( s_i \xleftarrow{\$} [0, \kappa) \), we have an SPACDP with ciphertexts \( c_i = m_i + s_i \kappa + r_ip \), and the approximate divisor \( p \) cannot be determined in polynomial time in the worst case. However, the offsets in this SPACDP are all of the form \( \mu_i + s_i \kappa \), for known \( m_i \), and we must make sure this does not provide information about \( p \). To show this, we rewrite the SPACDP as

\[
c_i = \mu_i + s_i \kappa + r_ip = \mu'_i + s'_i \kappa, \quad (i = 1, 2, \ldots, n), \tag{B.1}
\]

where \( s'_i = s_i + \lfloor (m_i + r_ip)/\kappa \rfloor \), and \( \mu'_i = \mu_i + r_ip \mod \kappa \). Now we may view \( [B.1] \) as an ACDP, with “encryptions” \( \mu'_i \) of the \( \mu_i \), and approximate divisor \( \kappa \). Since ACDP is at least as hard as SPACDP, and the offsets \( \mu'_i \) are polynomial time indistinguishable from uniform \([0, \kappa)\), from above, we will not be able to determine \( \kappa \) in polynomial time. Now, the offsets \( m'_1, m'_2 \) of any two plaintexts \( m_1, m_2 \) are polynomial time indistinguishable from \( m'_2, m'_1 \), since they are indistinguishable
from two independent samples from uniform \([0, \kappa)\). Therefore, in polynomial time, we will not be able to distinguish between the encryption \(c_1\) of \(m_1\) and the encryption \(c_2\) of \(m_2\).

**Theorem 4.** The encryption scheme produces ciphertexts with components which are random integers modulo \(pq\).

**Proof.** Consider a ciphertext vector which encrypts the plaintext, \(m\), and the expression \(m + rp + sa\) which represents one of its elements. Then \(r \xleftarrow{\$} [0, q)\), \(s \xleftarrow{\$} [0, pq)\).

Consider first \(m + sa\). We know that \(a^{-1} \mod pq\) exists because \(a \neq 0 \mod p\) and \(mod q\). Thus, conditional on \(r\),

\[
\Pr[m + rp + sa = i \mod pq] = \frac{1}{pq}.
\]

Since this holds for any \(i \in [0, pq)\), \(m + ra + sp \mod pq\) is a uniformly random integer from \([0, pq)\).

**Theorem 5.** If \(c\) is an encryption of \(m\) and \(c'\) is an encryption of \(m'\) then \(R(c \circ c') \mod pq\) is an encryption of \(mm'\).

**Proof.** Consider the Hadamard product modulo \(pq\), \(c \circ c'\), of the two augmented ciphertext vectors \(c_*\) and \(c'_*\):

\[
z_* = c_\circ c'_* = \begin{bmatrix} c_1c'_1 \\ c_2c'_2 \\ c_3c'_3 \end{bmatrix} \mod pq
\]

Therefore, if inputs \(m, m'\) are encrypted as \((m + rp) \mathbf{1} + sa\), \((m' + r'p) \mathbf{1} + s'a\), we first calculate

\[
z_* = (m + rp)(m' + r'p) \mathbf{1} + [(m + rp)s' + (m' + r'p)s]a_* + ss'a^{s2}_* \mod pq,
\]

where \(r_1 = mr' + m'r + rr'p\), \(s_1 = (m + rp)s' + (m' + r'p)s\), and \(a^{s2}_* = [a_1^2 a_2^2 a_3^2]^T\).
As we can see, \( z_* \) is not a valid encryption of \( mm' \). We need to re-encrypt this product to eliminate the \( a_1^{\circ 2} \) term.

We achieve this by multiplying \( z_* \) by \( R \). It is easy to check that \( Ra_* = a \), independently of \( a_1, a_2 \). Now

\[
(Ra_*^{\circ 2})_1 = (1 - 2a_1)a_1^2 + a_1a_2^2 + a_1(2a_1 - a_2)^2 \\
= a_1^2 + a_1((2a_1 - a_2)^2 + a_2^2 - 2a_1^2) \\
= a_1^2 + 2a_1(a_2 - a_1)^2 \\
= a_1^2 + a_1\beta \\
= \varphi p + \sigma a_1 \\
(Ra_*^{\circ 2})_2 = -2a_2a_1^2 + (a_2 + 1)a_2^2 + a_2(2a_1 - a_2)^2 \\
= a_2^2 + a_2((2a_1 - a_2)^2 + a_2^2 - 2a_1^2) \\
= a_2^2 + 2a_2(a_2 - a_1)^2 \\
= a_2^2 + a_2\beta \\
= \varphi p + \sigma a_2
\]

Thus, we obtain the identity \( Ra_*^{\circ 2} = \varphi p \mathbf{1} + \sigma \mathbf{a} \).

So, applying \( R \) to \( z_* \), i.e. \( z' = Rz_* \), gives

\[
z' = (mm' + r_1p)\mathbf{1} + s_1 Ra + ss' Ra^{\circ 2} = (mm' + r_1p)\mathbf{1} + s_1 a + ss' (\sigma a + \varphi p \mathbf{1}) \\
= (mm' + r_2p)\mathbf{1} + (s_1 + \sigma rr') a = (mm' + r_2p)\mathbf{1} + s_2 a \quad (\text{mod } pq)
\]

for some integers \( r_2, s_2 \). So \( z' \) is a valid encryption of \( mm' \).

**Theorem 6.** SPACDP is of equivalent complexity to the special case of HE2 where \( \delta = a_2 - a_1 \) (0 < \( \delta < p \)) is known.

**Proof.** Suppose we have a system of \( n \) approximate prime multiples, \( m_i + r_i p \) \( (i = 1, 2, \ldots, n) \). Then we generate values \( a, s_1, s_2, \ldots, s_n \overset{\$}{\leftarrow} [0, pq] \), and we have an oracle set up the cryptosystem with \( a_1 = a \), \( a_2 = a + \delta \). The oracle has access to \( p \) and provides us with \( R \), but no information about its choice of \( g \) and \( \sigma \). We
then generate the ciphertexts \( c_i \) \((i = 1, 2, \ldots, n)\):

\[
\begin{bmatrix}
    c_{i1} \\
    c_{i2}
\end{bmatrix} = \begin{bmatrix}
    m_i + r_i p + s_i a \\
    m_i + r_i p + s_i (a + \delta)
\end{bmatrix} \mod (pq). \tag{B.2}
\]

Thus \( c_{i1} - s_i a = c_{i2} - s_i (a + \delta) = m_i + r_i p \). Thus finding the \( m_i \) in (B.2) in polynomial time solves SPACDP in polynomial time.

Conversely, suppose we have any HE2 system with \( a_2 = a_1 + \delta \). The ciphertext for \( m_i \) \((i = 1, 2, \ldots, n)\) is as in (B.2). So \( s_i = \delta^{-1}(c_{i2} - c_{i1}) \). Since \( 0 < \delta < p < q \), \( \delta \) is coprime to both \( p \) and \( q \), and hence \( \delta^{-1} \mod pq \) exists. Thus breaking the system is equivalent to determining the \( m_i \mod p \) from \( m_i + \delta^{-1}(c_{i2} - c_{i1})a + r_i p \) \((i = 1, 2, \ldots, n)\). Determining the \( m_i + \delta^{-1}(c_{i2} - c_{i1})a \) from the \( m_i + \delta^{-1}(c_{i2} - c_{i1})a + r_i p \) \((i = 1, 2, \ldots, n)\) can be done using SPACDP. However, we still need to determine \( a \) in order to determine \( m_i \). This can be done by "deciphering" \( R \) using SPACDP.

We have

\[
2\delta^2 \alpha_1 = \sigma a - a^2 + gp, \quad 2\delta^2 \alpha_2 = \sigma(a + \delta) - (a + \delta)^2 + gp,
\]

so \( \sigma = 2\delta^2(\alpha_2 - \alpha_1) - 2\delta a - \delta^2 \). Now \( a \) can be determined by first determining \( m_0 = a(2\delta^2(\alpha_2 - \alpha_1) - (2\delta + 1)a - \delta^2) \) from \( m_0 + gp = 2\delta^2\alpha_1 \). This can be done using SPACDP. Then \( a \) can be determined by solving the quadratic equation \( m_0 = a(2\delta^2(\alpha_2 - \alpha_1) - (2\delta + 1)a - \delta^2) \mod p \) for \( a \). This can be done probabilistically in polynomial time using, for example, the algorithm of Berlekamp [42]. So the case \( a = [a \ a+\delta]^T \), with known \( \delta \), can be attacked using SPACDP on the system

\[
m_0 + gp, \ m_1 + \delta^{-1}(c_{i1} - c_{i2})a + r_1 p, \nonumber \\
\ldots, \ m_n + \delta^{-1}(c_{i1} - c_{i2})a + r_n p.
\]

\( \square \)

**Lemma 7.** \( \Pr(a_0, a_1, \ldots, a_{k-1} \text{ do not form a basis for } \mathbb{Z}_{pq}^k) \leq (k-1)(1/p + 1/q) \).

**Proof.** The \( a \)'s are a basis if \( A_k^{-1} \) exists, since then \( v = A_k r \) when \( r = A_k^{-1} v \), for any \( v \). Now \( A_k^{-1} \) exists mod \( pq \) if \((\det A_k)^{-1} \mod pq \) exists, by constructing the adjugate of \( A_k \). Now \((\det A_k)^{-1} \mod pq \) exists if \( \det A_k \neq 0 \mod p \) and \( \det A_k \neq 0 \mod q \). Now \( \det A_k \) is a polynomial of total degree \((k-1)\) in the \( a_{ij} \) \((0 < i \leq k, 0 < j < k)\), and is not identically zero, since \( \det A_k = 1 \) if...
\( a_i = e_{i+1} \ (1 < i < k) \). Also \( a_{ij} \overset{\text{d}}{\sim} [0,pq) \) implies \( a_{ij} \mod p \overset{\text{d}}{\sim} [0,p) \) and \( a_{ij} \mod q \overset{\text{d}}{\sim} [0,q) \). Hence, using the Schwartz-Zippel Lemma (SZL) \([236]\), we have \( \Pr(\det A_k = 0 \mod p) \leq (k-1)/p \) and \( \Pr(\det A_k = 0 \mod q) \leq (k-1)/q \), and it follows that \( \Pr(\det A_k^{-1} \mod pq) \leq (k-1)(1/p + 1/q) \).

**Theorem 9.** \( A_{*k}^{2} \) has no inverse \( \bmod pq \) with probability at most \( (k^2-1)(1/p + 1/q) \).

**Proof.** We use the same approach as in Lemma\(\text{[7]}\). Thus \( A_{*k}^{2} \) is invertible provided \( \det A_{*k}^{2} \neq 0 \bmod p \) and \( \det A_{*k}^{2} \neq 0 \bmod q \). Let \( A \) denote the vector of \( a_{ij} \)'s, \( (a_{ij} : 1 \leq i \leq k, 1 \leq j < k) \). The elements of \( A_{*k}^{2} \) are quadratic polynomials over \( \mathbb{A} \), except for the first column, which has all 1’s, and columns 2, 3, \ldots, \( k \) which are linear polynomials. So \( \det A_{*k}^{2} \) is a polynomial over \( \mathbb{A} \) of total degree \( 2(\frac{k}{2}) + k-1 = k^2 - 1 \). Thus, unless \( \det A_{*k}^{2} \) is identically zero as a polynomial over \( \mathbb{A} \), the SZL \([236]\) implies \( \Pr(\det A_{*k}^{2}^{-1} \bmod p) \leq (k^2-1)/p \) and \( \Pr(\det A_{*k}^{2}^{-1} \bmod q) \leq (k^2-1)/q \). Therefore we have \( \Pr(\det A_{*k}^{2}^{-1} \bmod pq) \leq (k^2-1)(1/p + 1/q) \).

It remains to prove that \( \det A_{*k}^{2} \) is not identically zero as a polynomial over \( \mathbb{A} \) in either \( \mathbb{Z}_p \) or \( \mathbb{Z}_q \). We prove this by induction on \( k \). Consider \( \mathbb{Z}_p \), the argument for \( \mathbb{Z}_q \) being identical. Since \( \mathbb{Z}_p \) is a field, \( \det A_{*k}^{2} \) is identically zero if and only if it has rank less than \( \binom{k+1}{2} \) for all \( A \). That is, there exist \( \lambda_{ij}(A) \in \mathbb{Z}_p \) \( (0 \leq i \leq j < k) \), not all zero, so that

\[
\mathcal{L}(A) = \sum_{0 \leq i \leq j}^{k-1} \lambda_{ij} a_{*i} \circ a_{*j} = \alpha + a_{*k-1} \circ \beta + \lambda_{k-1,k-1} a_{*k-1}^{2} = 0,
\]

where \( \alpha = \sum_{0 \leq i \leq j}^{k-2} \lambda_{ij} a_{*i} \circ a_{*j} \) and \( \beta = \sum_{i=0}^{k-2} \lambda_{i,k-1} a_{*i} \) are independent of \( a_{*k-1} \).

Clearly \( \lambda_{k-1,k-1} = 0 \). Otherwise, whatever \( \alpha, \beta \), we can choose values for \( a_k \) so that \( \mathcal{L} \neq 0 \), a contradiction. Now suppose \( \lambda_{i,k-1} \neq 0 \) for some \( 0 \leq i < k-1 \). The matrix \( \hat{A} \), with columns \( a_{*i} \) \( (0 \leq i < k-1) \) contains \( A_{k-1} \) as a submatrix, which has rank \( (k-1) \) with high probability by Lemma\(\text{[7]}\). Thus \( \beta \neq 0 \) and, whatever \( \alpha \), we can choose values for \( a_k \) so that \( \mathcal{L} \neq 0 \). Thus \( \lambda_{i,k-1} = 0 \) for all \( 0 \leq i < k \). Thus \( \lambda_{ij} \neq 0 \) for some \( 0 \leq i \leq j < k-1 \). Now the matrix \( \hat{A}_{*i}^{2} \) with \( \binom{k}{2} \) columns \( a_{*i} \circ a_{*j} \) \( (0 \leq i \leq j < k-1) \) contains \( A_{*k-1}^{2} \) as a submatrix, and therefore has rank \( \binom{k}{2} \) by induction. Hence \( \alpha \neq 0 \), implying \( \mathcal{L} \neq 0 \), a contradiction. \( \square \)
Lemma 8. Let \( A_{*k} = [a_{*0} a_{*1} \ldots a_{*k-1}] \), where the columns of \( A_k \) form a basis for \( \mathbb{Z}_{pq}^k \). If \( RA_{*k} = A_k \), then \( Rv_\ast = v \) for all \( v \in \mathbb{Z}_{pq}^k \).

Proof. We have \( v = A_k r \) for some \( r \in \mathbb{Z}_{pq}^k \). Then \( A_{*k} = U_k A_k \) and \( v_{*k} = U_k v \), so \( Rv_\ast = RU_k v = RU_k A_k r = RA_{*k} r = A_k r = v \).

Theorem 10. If \( c \) is an encryption of \( m \) and \( c' \) is an encryption of \( m' \) then \( R(c_\ast \circ c'_\ast) \) (mod \( pq \)) is an encryption of \( mm' \).

Proof. Consider the Hadamard product of two augmented ciphertext vectors, \( c_\ast \circ c'_\ast \). For notational brevity, let \( \tilde{m} = m + rp \).

\[
\begin{align*}
c_\ast \circ c'_\ast &= (\tilde{m}_1 \ast + \sum_{j=1}^{k-1} s_j a_{*j}) \ast (\tilde{m'}_1 \ast + \sum_{j=1}^{k-1} s'_j a_{*j}) \\
&= \tilde{m}\tilde{m'}_1 \ast + \sum_{j=1}^{k-1} (\tilde{m}s'_j + \tilde{m'}s_j) a_{*j} + \sum_{j=1}^{k-1} s'_ja_{*j} \ast a_{*j} \\
&+ \sum_i 1 \leq i < j \leq k-1 (s_is'_j + sjs_j) a_{*i} \ast a_{*j},
\end{align*}
\]

since \( 1 \ast v_\ast = v_\ast \) for any \( v \). There are \( \binom{k}{2} \) product vectors, which we must eliminate using the re-encryption matrix \( R \), a \( k \times \binom{k+1}{2} \) matrix.

From Lemma 8, we have that \( Rv_{*k} = v \) for all \( v \in \mathbb{Z}_{pq}^k \). Therefore, \( RA_{*k} = A_k \). However, this condition can be written more simply, since it is \( RU_k A_k = A_k \). Postmultiplying by \( A_k^{-1} \) gives \( RU_k = I_k \). For example, if \( k = 3 \),

\[
\begin{bmatrix}
R_{11} & R_{12} & R_{13} & R_{14} & R_{15} & R_{16} \\
R_{21} & R_{22} & R_{23} & R_{24} & R_{25} & R_{26} \\
R_{31} & R_{32} & R_{33} & R_{34} & R_{35} & R_{36}
\end{bmatrix}
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1 \\
2 & -1 & 0 \\
2 & 0 & -1 \\
0 & 2 & -1
\end{bmatrix}
= \begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0 \\
0 & 0 & 1
\end{bmatrix}.
\]

So \( R_{11} + 2R_{14} + 2R_{15} = 1 \), \( R_{11} = 1 - 2R_{14} - 2R_{15} \), etc. Now, since \( RA_{*k} = A_k \), we have

\[
R(c_\ast \circ c'_\ast) = (mm' + \hat{r}p)1 + \sum_{j=1}^{k-1} \hat{s}_ja_j + \sum_{1 \leq i < j \leq k-1} \hat{s}_{ij} R(a_{*i} \ast a_{*j}),
\]

where \( \hat{r} \), \( \hat{s}_j \) and \( \hat{s}_{ij} \) \((1 \leq i < j \leq k - 1)\) are some integers.

There are \( k\binom{k}{2} \) undetermined parameters \( R_{it} \), \( 1 \leq i \leq k \), \( k < \ell \leq \binom{k+1}{2} \). We now determine these by setting

\[
R(a_{*i} \ast a_{*j}) = \sigma_{ij} p 1 + \sum_{l=1}^{k-1} \sigma_{ijkl} a_l
\]
Thus we have $k^2$ new unknowns, the $\varrho$'s and $\sigma$'s, and $k^2$ linear equations for the $k^2$ unassigned $R_{i\ell}$'s. Let $A_{ik}^{2k}$ be the $(k^2+1) \times (k^2+1)$ matrix with columns $a_{i}\circ a_{j}$ ($0 \leq i < j < k$), and let $C_k$ be the $k \times (k^2)$ matrix with columns $\varrho_{ij}p^1 + \sum_{l=1}^{k-1} \sigma_{ijl} a_l$ ($0 < i < j < k$). Then the equations for the $R_{i\ell}$ can be written as

$$RA_{ik}^{2k} = [A_k \mid C_k].$$

which by construction of $R$ (Algorithm 7.21) has a solution. 

**Lemma 11.** Pr(det $C = 0$ mod $pq) \leq (2k - 1)(1/p + 1/q)$.

**Proof.** From Lemma 7 det $A = 0$ mod $p$ or det $A = 0$ mod $q$ with probability at most $(k - 1)(1/p + 1/q)$. So det $A$ is not zero or a divisor of zero mod $pq$. The entries of $W'$ are random $[0, pq)$, and det $W'$ is a polynomial of total degree $k$ in its entries. It is a nonzero polynomial, since $W' = I_k$ is possible. Hence, using the SZL [236], Pr(det $W' = 0$ mod $p) \leq k/p$ and Pr(det $W' = 0$ mod $q) \leq k/q$. So det $W'$ is zero or a divisor of zero mod $pq$ with probability at most $k(1/p + 1/q)$. So det $A$ det $W' = 0$ mod $pq$ with probability at most $(2k - 1)(1/p + 1/q)$. So det $C \neq 0$ with high probability.

**Lemma 12.** Pr(det $C_0 = 0$ mod $pq) \leq (2k - 1)(1/p + 1/q)$.

**Proof.** Note that $C_0 = C$ if $m_1 = m_2 = \cdots = m_k = 0$. Since Lemma 11 holds in that case, the result follows.

**Lemma 13.** Equation (7.6) holds if and only if $\sum_{t=1}^{k-1} \sigma_{jlt} \varrho_{lt} \equiv \sum_{t=1}^{k-1} \sigma_{ijt} \varrho_{lt}$ (mod $q$), $\forall i,j,l \in [1,k-1]$.

**Proof.** Since $\gamma^T 1 = 1$ and $\gamma^T a_i = 0$, $i \in [1,k-1]$, $\gamma^T (a_i \cdot a_j) = \gamma^T (p\varrho_{ij} 1 + \sum_{t=1}^{k-1} \sigma_{ijt} a_t) = p\varrho_{ij}$ . Thus

$$a_i \cdot (a_j \cdot a_i) = a_i \cdot (p\varrho_{ij} 1 + \sum_{t=1}^{k-1} \sigma_{ijt} a_t) = p\varrho_{ij} a_i + \sum_{t=1}^{k-1} \sigma_{ijt} a_i \cdot a_t,$$

and hence $\gamma^T [a_i \cdot (a_j \cdot a_i)] = p\sum_{t=1}^{k-1} \sigma_{ijt} \varrho_{lt}$. Similarly $\gamma^T [(a_i \cdot a_j) \cdot a_i] = p\sum_{t=1}^{k-1} \sigma_{ijt} \varrho_{lt}$, and the lemma follows.
Lemma 14. Let $\tau, \varrho_i \sim \mathcal{U}[0, q)$ ($i \in [1, k - 1]$), let $\varrho_{ij} = \varrho_i \varrho_j \mod q$, and let the $\sigma_{ijt}$ satisfy $\sum_{t=1}^{k-1} \sigma_{ijt} \varrho_t = \tau \varrho_i \varrho_j \mod q$ for all $i, j \in [1, k - 1]$. Then, for all $i, j, \ell \in [1, k - 1]$, $\gamma^T(a_i \cdot (a_j \cdot a_\ell)) = \tau \varrho_i \varrho_j \varrho_\ell \mod q$, the symmetry of which implies (7.6).

Proof. We have $\gamma^T(a_j \cdot a_\ell) = p \varrho_{ij} = p \varrho_j \varrho_\ell$ for all $j, \ell \in [1, k - 1]$. Hence, mod $q$,

$$
\gamma^T(a_i \cdot (a_j \cdot a_\ell)) = p \sum_{t=1}^{k-1} \sigma_{jjt} \varrho_t,
\quad = p \sum_{t=1}^{k-1} \sigma_{jjt} \varrho_t \\
\quad = p \varrho_i \sum_{t=1}^{k-1} \sigma_{jjt} \varrho_t \\
\quad = p \varrho_i \tau \varrho_j \varrho_\ell = p \tau \varrho_i \varrho_j \varrho_\ell.
$$

\hfill \Box

B.2 Chapter 8

Theorem 15. The increase in bit leakage for $m$ as a result of this flattening is approximately $\lg(mp_m/F(m))$, where $p_m = F(m) - F(m-1)$.

Proof. We will assume that $F(m)$ is a reasonably smooth distribution, so $F'(m)$ exists, and is approximately equal to the frequency function $p_m = F(m) - F(m-1)$. We have shown that $\tilde{m} = NF(m)$ is approximately uniform on $[0, N]$. Also, we have shown that we can estimate $\tilde{m}$ from $c(\tilde{m})$ only to within $\tilde{r} = \tilde{m}/n = NF(m)/n$. Thus we can estimate $m$ to within $r$, where

$$
NF(m)/n \approx NF(m + r) - NF(m) \approx rNF'(m) \approx rNP_m,
$$

and hence $r \approx F(m)/np_m$. Thus the bit leakage is

$$
\lg m - \lg(F(m)/np_m) = \lg n + \lg(mp_m/F(m)).
$$

Thus the increase in bit leakage for $m$ is approximately $\lg(mp_m/F(m))$. \hfill \Box
B.3 Chapter 9

Theorem 16. If $SE$ is IND-CPA secure, $f$ is a secure pseudorandom function, then $SSE$ is UP-IND-CPA.

Proof. A receives the pair $(c,t)$ and must determine whether $(c,t) = (c_0,t_0)$ or $(c,t) = (c_1,t_1)$. A might try to distinguish $c_0, c_1$, knowing $m_0, m_1$. Let $\varepsilon_E$ be A’s advantage for doing this. Then $\varepsilon_E$ is negligible, since $SE$ is IND-CPA secure.

So A can only win by determining that the tag $t$ matches either $m_0$ or $m_1$. To do this, A must distinguish $f(m_0)$ from $f(m_1)$. Let $K' = K_M \setminus f(Q)$, the set of possible values of $f(m_0), f(m_1)$. Let $E(K_0,K_1)$ be the event that $\{f(m_0), f(m_1)\} = \{K_0, K_1\}$, for $K_0, K_1 \in K'$. Then, since $f$ is a secure pseudorandom function and $f(m_0), f(m_1)$ have not been observed, we have

$$\Pr(f(m_0) = K_0, f(m_1) = K_1 \mid E) = \Pr(f(m_0) = K_1, f(m_1) = K_0 \mid E) = \frac{1}{2} + \varepsilon_f,$$

where $\varepsilon_f$ is negligible. Thus $\varepsilon_f$ bounds A’s negligible advantage for distinguishing $f(m_0)$ from $f(m_1)$ without using the tag $t$. Now suppose A does use the tag $t$ to try to distinguish between $f(m_0)$ and $f(m_1)$. Suppose that A could even compute the MAC key $K_M$ for $c$ knowing only that $t = M(K_m, c)$. Then, since $c_0, c_1$ are distinguishable only with advantage $\varepsilon_E$, A will know only $K_M = f(m_0)$ or $K_M = f(m_1)$, but will only know which with advantage at most $\varepsilon_f$. So A’s advantage for achieving $d = b$ is at most $\varepsilon_E + \varepsilon_f$, which is negligible.

Theorem 17. Let $E$ be a sufficiently random block cipher with $B = |C|$. Let $D$ be the size of the dictionary of valid $m \in M$, let $T = |T|$ and let $c_m = E(k_E, m)$. Then

$$\Pr[\exists m, m' \in M, m \neq m' : M(c_m, f(m)) = M(c_{m'}, f(m'))] \leq \frac{1}{BT} + \left(1 - \frac{1}{B}\right) \frac{D^2}{2T}.$$

Proof. From the randomness of $E$, we have

$$\Pr[\exists m, m', m \neq m' : E(k, m) = E(k, m')] = 1/B.$$
Therefore,

$$\Pr[\exists m, m' \in \mathcal{M}, m \neq m' : M(c_m, f(m)) = M(c_{m'}, f(m))] \leq \frac{1}{BT} + \left(1 - \frac{1}{B}\right) \frac{D(D - 1)}{2T} \times \frac{1}{T}$$

$$\leq \frac{1}{BT} + \left(1 - \frac{1}{B}\right) \frac{D^2}{2T}$$

Given that $B$ is large enough and $D^2 \ll T$ this probability is negligible.  \qed
Appendix C

Derivation of Bounds

To recap, \( n \) is the number of inputs, \( M \) is an exclusive upper bound on the inputs, \( d \) is the degree of the polynomial we wish to calculate. We take \( p \approx 2^\lambda \) and then \( q \approx 2^n \), where \( \eta = \lambda^2 / \rho - \lambda \), to guard against the attacks of [92, 164].

For HE1, we assume \( M \approx 2^\rho \), \( n \leq \sqrt{M} \). Therefore,

\[
p > (n + 1)^d M^d \approx (nM)^d \text{ for large } n.
\]

So, we may take

\[
p = 2^\lambda > M^{3d/2} \approx 2^{3d\rho/2}
\]

i.e. \( \lambda \approx 3d\rho/2 \)

and

\[
\eta \approx \frac{\lambda^2}{\rho} - \lambda = \frac{3d\lambda}{2} - \lambda = \frac{3d\rho}{2} \left( \frac{3d}{2} - 1 \right)
\]

For HE1N, we assume \( M \approx 2^\rho \), and we have \( \rho' = \rho + \lg \kappa \). Now,

\[
\kappa > (n + 1)^d M^d \approx (nM)^d \text{ for large } n,
\]

i.e. \( \lg \kappa \approx d(\lg n + \rho) \)
Therefore, since $\rho = \rho' - \lg \kappa,$

$$\lg \kappa > d \lg n + d(\rho' - \lg \kappa)$$
i.e. $\lg \kappa \approx \frac{d(\lg n + \rho')}{d + 1}$

Since $\kappa$ is much larger than $M$, we also have

$$p = 2^\lambda > (n + 1)^d(M + \kappa^2)^d \approx (n\kappa^2)^d \text{ for large } n$$
i.e. $\lambda \approx d(\lg n + 2 \lg \kappa)$,
and $\eta \approx \frac{\lambda^2}{\rho'} - \lambda = \frac{3d\lambda}{2} - \lambda = \frac{3d\rho'}{2} \left( \frac{3d}{2} - 1 \right)$

Then we can calculate $\eta$ as for HE1 above. Note that, in both HE1 and HE1N, $\lambda$ scales linearly with $d$, and $\eta$ scales quadratically. These bounds carry over to HE2, HE2N, HE$k$ and HE$k$N.
Glossary

**block** In a block-oriented file system, the smallest unit of storage is a block. Each file in the system is stored as several blocks. In GFS, a block is called a chunk.

**cluster** A collection of nodes operating in concert to execute a parallel application.

**framework** The MapReduce run-time system that executes the user-defined code as part of a job.

**input split** A logical division of the input. There will be at least as many input splits as there are map tasks. Each map task is assigned at least one input split to process.

**intermediate data** Data output by a map task.

**job** A job is a MapReduce computation. A job is composed of several tasks.

**lagging** Executing considerably slower than similar processes.

**mapper** A worker process executing a map task.

**map phase** The phase of the computation when the map tasks are executed.

**map task** A map task comprises the framework code to perform the following sequence: it reads the input and formats it as key-value pairs; invokes the user-defined map function on each key-value pair; and writes the output of the map function into several partitions.
**node** A node is a machine that executes a worker process.

**partition** The output of each map task is divided into a number of sets of data. There will be as many sets as reduce tasks. Each set is called a partition.

**reduce phase** The phase of the computation when the reduce tasks are executed.

**reducer** A worker process executing a reduce task.

**reduce task** A reduce task comprises the framework code to perform the following sequence: it reads the relevant intermediate data partition from each map worker; it groups the input key-value pairs by key; it invokes the user-defined reduce function on each key-value list pair; and writes the output of the reduce function to storage.

**shuffle phase** The phase of the computation where the intermediate data produced by each map task is collected per partition and distributed to the appropriate reduce task. Each partition of data is sorted and grouped by key during the shuffle phase. Grouped pairs are merged into a single key-value pair.

**task** A component of the application that is executed in parallel by a worker process.