PROVABLE DATA POSSESSION USING SIGMA PROTOCOLS

A Thesis
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Akshaya Mohan

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Electrical and Computer Engineering

March 2013

Fargo, North Dakota

North Dakota State University
Graduate School

Title

Provable Data Possession using Sigma protocols

By

Akshaya Mohan

The Supervisory Committee certifies that this ***disquisition*** complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Rajendra Katti
Chair

Dr. Sudarshan Srinivasan

Dr. Kendall E. Nygard

Approved by Department Chair:

04/02/2013
Date

Dr. Rajendra Katti
Department Chair

ABSTRACT

A Provable Data Possession (PDP) scheme allows a client which has stored data at an untrusted server to verify that the server possesses the original data that it stored without retrieving the entire file. In this thesis study, a new PDP scheme is built using the concept of sigma protocols. The client pre-processes a file and stores it on the server. At a later time, the client issues a challenge to the server requesting it to compute a Proof of Possession. The client verifies the response using its locally stored metadata. The challenge-response protocol that is derived from the sigma protocol, minimizes both computation and communication complexity.

Implementation and complexity analysis of the algorithms used in the ∑-PDP scheme was done as a part of this thesis.

The main goal of this research was to minimize computation and communication complexity of ∑-PDP scheme as compared to the existing PDP schemes.

ACKNOWLEDGEMENTS

First of all, I would like to express my sincere gratitude to my advisor, Dr. Rajendra Katti, for his continuous interest, patience, and guidance throughout the research and in preparation of this thesis.

I wish to thank the rest of my supervisory committee: Dr. Sudarshan Srinivasan, and Dr. Kendall E. Nygard, for their encouragement and supervising my final exam.

I would also like to thank my parents and brother on whose support and love I have relied throughout my Masters.

Last but not the least; I would like to thank my friends for their encouragement and support.

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGEMENTS iv

LIST OF TABLES viii

LIST OF FIGURES ix

CHAPTER 1. INTRODUCTION 1

1.1. Background 1

1.2. Motivation 2

1.3. Research Objective 3

1.4. Thesis Outline 5

CHAPTER 2. LITERATURE REVIEW 7

2.1. Related Work 7

2.1.1. Comparison of PDP and POR 7

2.1.2. Various PDP schemes 9

2.2. Provable Data Possession (PDP) 10

CHAPTER 3. BASIC CRYPTOGRAPHIC PRIMITIVES 11

3.1. Public-key Encryption 11

3.2. Zero-knowledge Proofs 11

3.3. Proofs of Knowledge 11

3.4. Properties of Completeness and Soundness 11

3.4.1. Completeness 12

3.4.2. Soundness 12

3.5. Sigma Protocol [15] 12

3.6. Okamoto Protocol 16

3.6.1. Properties of witness-indistinguishability and witness-hiding 17

3.7. One-time Signature Scheme 18

3.8. Pseudorandom Function (PRF) 20

3.9. Pseudorandom Permutation (PRP) 20

3.10. Pseudorandom-Generator 21

CHAPTER 4. ∑-PROVABLE DATA POSSESSION 22

4.1. Details of the Proposed ∑-PDP Scheme 22

4.1.1. $KeyGen(1^k)$ 23

4.1.2. $Sign(sk, m_i)$ 23

4.1.3. $Ver(pk_i, m_i, \sigma_{i,m_i}))$ 24

4.1.4. $GenProof(\ F = \ (m_1\ , m_2\ , \ldots, m_n\ ), e, \sum\ = \ (\sigma_{1,m_1}\ , \sigma_{2,m_2}\ , \ldots, \sigma_{n,m_n}\ ))$ 24

4.1.5. $CheckProof(pk, sk, e, (M, (s_1, s_2)))$ 25

4.2. ∑-Provable Data Possession 26

4.3. ∑-PDP Scheme 27

4.4. Data Possession Game 28

4.5. Proof of Security 30

4.6. Complexity Analysis and Comparison 34
4.6.1. Computations to generate tags/signatures at the client (offline computations) 38
4.6.2. Computations to generate proof at server (online computations) 39
4.6.3. Computations to check proof at the client on c challenge blocks 41
4.6.4. Comparison of ∑-PDP with previous work 44
4.6.5. Computations involved in ∑-PDP as compared to POS 44
CHAPTER 5. IMPLEMENTATION 46
5.1. Timing Analysis for Varying Message and Challenge Lengths 47
CHAPTER 6. CONCLUSION AND FUTURE RESEARCH 49
6.1. Main Contribution and Results 49
6.2. Limitations and Future Research 50
BIBLIOGRAPHY 51
APPENDIX A. SOURCE CODE 56
A.1. Code to Generate Witnesses, Secret Parameters and Input Message Blocks 56
A.2. Code to Generate Signatures with Modular Exponentiation Multiplication Function 58
A.3. Code to Generate Challenge and Proof on this Challenge 58
A.4. Code to Check Proof 59

LIST OF TABLES

Table Page

1. Computations involved in S-PDP and ∑-PDP 35

2. Modular multiplications involved in ∑-PDP in comparison with S-PDP 37

3. Multiplications to generate tags/signatures at client 38

4. Multiplications to generate proof at server 40

5. Online multiplications to check proof at the client 41

6. Offline multiplications to check proof at the client 43

7. Time to perform computations for varying file and challenge length 48

LIST OF FIGURES

Figure Page

1. Protocol representation of ∑-protocol template 14

2. Protocol representation of Okamoto-protocol 17

3. ∑-PDP scheme 28

4. Multiplications at client to generate tags/signatures 39

5. Multiplications at the server to generate proof 40

6. Online multiplications at client to check proof 42

7. Offline computations to check proof at client 43

CHAPTER 1. INTRODUCTION

This chapter gives the introduction for this thesis study. This includes background, motivation and research objective of this thesis study.

1.1. Background

Cloud computing has been gaining significant momentum in recent years. [1] Advances in networking technology and the rapid accumulation of information have fueled a trend towards outsourcing data management to external service providers. As suggested by [2], cloud computing frees organizations from the need to buy and maintain their own hardware and software infrastructure. By outsourcing, organizations and individuals can concentrate on their core tasks rather than incurring the substantial hardware, software and personnel costs involved in maintaining data "in house". The main problem is in verifying that the server continually and faithfully stores the entire and authentic content entrusted to it by the client. So, the server that stores the client's data is not necessarily trusted. The server is untrusted in terms of both security and reliability: it might maliciously or accidentally erase the data or place it onto slow or offline storage media. This could occur for numerous reasons, including to save storage or to comply with external pressures. Also, the server could accidentally erase some data and choose not to notify the owner. Therefore, users might want to check if their data has been tampered with or deleted. However, as in [3] outsourcing the storage of very large files to a remote server presents an additional constraint: the client should not download all stored data in order to verify it since the bandwidth and time available may be prohibitive, especially if the client performs this check frequently.

We note that traditional approaches to verify integrity, such as message authentication codes or digital signatures, cannot be applied since the client does not store the data. To address

this, a model called a Provable Data Possession (PDP) was first put forth by Ateniese [4]. A PDP system is a proof system based on public-key techniques that enables the server to efficiently prove to the client or anyone in possession of the client's public-key that it possesses the client's data. According to [1], PDP systems are similar to proofs of knowledge (POK) ([5], [6], [7]) which are proof systems that enable a prover to convince a verifier that it knows a secret in "zero-knowledge", i.e., without leaking any partial information about the secret to the verifier. These schemes provide probabilistic guarantees of possession, where the client checks a random subset of stored blocks with each challenge that it generates.

1.2. Motivation

While cloud computing provides many benefits, it also brings in new challenges in research. One of the main challenges is to achieve efficiency in communication, storage and computation on both client and server sides. As Whitfield Diffie [8] said, "The whole point of cloud computing is economy". The PDP scheme of [4] has a large number of exponentiations involved in generating the proof of possession. Exponentiations can be expensive and therefore any such computation at the server is eliminated in the proposed model. Also, the number of exponentiations are fixed to 5 at the client making it independent of the size of the challenge. Thus, the proposed scheme is more efficient. Even though the scheme in [4] and the proposed method have a computational complexity of $O(1)$, the proposed method has a major reduction in the number of exponentiations. This model was found to be useful in an application that stores a file from a hard drive onto a computer system. The storage-retrieval of this file can be run using the challenge-response protocol presented in this thesis study. The reliable storage of this file can be efficiently checked by challenging the computer system of the file that was stored on it. The

party storing the file can be convinced of storage on checking for the proof of possession that the computer sends it.

The main goal of this research was to develop a provably-secure probabilistic model of PDP that is more efficient than previous solutions in terms of computation and communication complexity at both the client and the server of the PDP system. This in turn would reduce network communication.

1.3. Research Objective

In this thesis, we focus on archival network storage. Given that the file is large and is stored at a remote site, accessing the entire file is expensive. Therefore, to increase scalability and limit bandwidth, a ∑-protocol model is defined for provable data possession that allows the client to be able to verify that a server has stored a file without retrieving the data from the server and without having to access the entire file. This allows the server to access small portions of the file in generating a proof. A provably-secure scheme for PDP using ∑-protocols is formalized in this thesis study. The client stores a small amount of data (public and secret key) to verify the server's proof. The scheme uses $O(1)$ bandwidth. Each challenge requires small, constant amount of communication between client and server. In order to generate the proof, the server accesses c blocks where c is a small constant and a subset of n where n is the input file size and is about 1,000,000 blocks in length. In terms of computation, there are modular exponentiations, multiplications and additions involved in the proof of possession. Since the size of the file is n, the storage at the server is $O(n)$. The client computes one-time signatures on each file block and then stores the file along with the corresponding signatures with the server. At a later time, the client can verify if the server possesses the file by sending a random challenge against a randomly selected set of file blocks. Using the queried blocks and their corresponding signatures,

the server generates a proof of possession. The proof of possession is a linear combination of a constant number c of file blocks and a linear combination of their corresponding signatures. The client checks the validity of the proof with the help of secret data that it stored and is thus convinced of data possession, without actually having to retrieve the file blocks. The proof of security is based on the discrete logarithm assumption.

The proposed scheme can be easily understood by first considering ∑-protocols. A ∑-protocol is a three-move protocol between a prover and a verifier. The prover proves to the verifier that it knows a witness to a common input without revealing the witness. This is called Zero-Knowledge Proof of Knowledge (ZKPOK). One-time signatures are generated based on this simple protocol using the Okamoto protocol, an example of a ∑-protocol. The signing key in the one-time signature scheme consists of a pair of witness and a pair of secret parameters. These secret parameters are randomly chosen each time a new message has to be signed. Therefore, a new signing key is used to sign every new message, making the use of one-time signatures possible. The prover also computes the public key for each message block which when given to the verifier along with the signature gives provision to check the validity of the signature. The client acts as the prover and the server behaves as the verifier of the signatures that the client generates.

A random challenge chosen by the client consists of indices and co-efficients. To generate a proof of possession for this challenge, the server has to possess the blocks indicated by the indices to multiply them with the corresponding co-efficients and send across a small amount of data to the client for verification. The client needs to have the private keys to verify this proof. Therefore, the client is convinced of data possession, without actually having to

retrieve the file blocks, hence minimizing bandwidth. The security property of the proposed scheme is proved under standard assumptions.

The goal of this thesis is to construct an efficient PDP scheme for proofs of storage using ∑-protocols. Our main strategy is to devise an efficient protocol using sigma protocols and one-time signatures for PDP systems.

We refer to the complexity parameters throughout this thesis. The definition of the performance parameters of ∑-PDP follows the parameters from [4]. They include:

- Computation complexity: The computational cost to pre-process a file F (at C), to generate a proof of possession (at S) and to verify this proof (at C);
- Block access complexity: The number of file blocks accessed to generate a proof of possession (at S);
- Communication complexity: The amount of data transferred (between C and S).

1.4. Thesis Outline

This thesis is organized as follows: Chapter 2 gives an overview of related work. Chapter 3 gives a detailed description of Provable Data Possession. It also outlines ∑-protocols and its properties, followed by the definition of the Okamoto protocol that provides basis for the proposed scheme. A one-time signature scheme based on the Okamoto protocol is also presented.

Chapter 4 gives the definitions for ∑-PDP schemes followed by the construction of the proposed scheme (∑-PDP). This chapter also gives the details of the complexity of the scheme. The scheme is compared to previous work in terms of exponentiations being represented as

multiplications. This section also gives details of comparison of PDP with POS. Chapter 5 presents the implementation details of the algorithms used in the proposed scheme using Crypto++. Finally, Chapter 6 includes the conclusions derived from the study and recommendations for future research.

CHAPTER 2. LITERATURE REVIEW

This section summarizes related work in PDP systems and introduces the concept of Provable Data Possession.

2.1. Related Work

This section consists of comparison of PDP and POR. It also has the details of various PDP schemes. The Provable Data Possession is given later in this section.

2.1.1. Comparison of PDP and POR

At the first glance, proving possession of data might seem like a simple problem which can be solved using known cryptographic primitives like collision-resistant hash functions. As an example, the owner could store the hash of the data and ask the server to return a copy of the data so that it can verify its integrity. Unfortunately, there are many practical settings in which such a simple approach will not work. If the bandwidth available between the client and the server is limited, then transferring large amounts of data might be infeasible. Also, if the client is computationally limited, then it might not be able to process the entire data collection. Finally, if the owner has limited storage-which, after all, is the motivation for outsourcing storage to begin with-then it might be impossible for it to maintain a copy of the data for verification purposes. Recently, two approaches to the problem of proving possession of data were put forth ([4], [7]).

While all these works address the same problem, the approaches and techniques used are very different. The first approach, referred to as a proof of retrievability (POR), is based on symmetric-key techniques and enables a server to prove to an owner that it possesses enough of the original data to allow for its efficient retrievability (i.e., in polynomial-time). On the other hand, the second approach, referred to as a proof of data possession (PDP), is based on public-

key techniques and allows a server to prove to anyone with the owner's public-key that it possesses the original file. Both PORs and PDPs are proof systems executed between a prover and a verifier that enables the prover to convince the verifier that he is in possession of a file F. They are generally composed of two phases: a setup phase where the owner of the file encodes the file and sends it to the prover; and a challenge phase where a verifier engages in an interactive protocol with the prover to determine if it indeed possesses the file. If only the owner is allowed to verify possession, then the system is privately verifiable. If the verifier can be any party that possesses the owner's public-key then the system is publicly verifiable. If the verifier is willing to store a commitment of F, then the prover can execute a zero-knowledge proof of knowledge (POK) for F. A POK allows a prover to convince a verifier that is in possession of a commitment, that it knows the secret under the commitment. In addition, this is done while guaranteeing that the verifier does not learn any information about the secret. For our purposes, however, where files can be very large, we are interested in proof systems with communication complexity and storage at the verifier that is $O(1)$.

In this thesis, we present an explicit connection between POK and PDP systems. Concretely, we show how to compile certain 3-move POK into PDP systems with $O(1)$ size proofs. Due to their efficiency these protocols are widely used in practice and many efficient constructions are known. We use our compiler on an example $\sum$-protocol for proving knowledge of discrete logarithms [9] to generate a privately verifiable PDP system. The resulting construction is the first $\sum$-protocol construction based on the hardness of computing discrete logarithms. It is efficient and supports an unlimited number of proofs.

2.1.2. Various PDP schemes

Ateniese *et al*. [4] were the first to have formalized the PDP model. The authors present several variations of their scheme under different cryptographic assumptions. These schemes provide probabilistic guarantees of possession, where the client checks a random subset of stored blocks with each challenge. We refer to [4] for a detailed review on earlier relevant works.

The PDP scheme given in [4] provides an optimal protocol for the static case. Reference [3] is an improved scheme of the original PDP in that the client asks the server to prove that the stored data has not been tampered with or deleted. In this paper, they propose constructions for dynamic provable data possession, which extends the PDP model to support provable updates on the stored data. Pietro *et al*. [10] gave another dynamic PDP solution called Scalable PDP. This PDP technique is based on symmetric key cryptography (without requiring any bulk encryption), it also supports operations such as modification, deletion and append. This is known to be a very efficient PDP scheme for operations that are only append-like. Reference [11] uses the PDP framework for remote data checking based on spot checking where in corruption of any fraction of the data stored at the server can be detected by the client. They integrate forward error-correcting codes into PDP. Reference [12] gives a PDP scheme in hybrid clouds that supports dynamic scalability in which they consider multiple cloud service providers to cooperatively store and maintain the client's data. Another extension of the original PDP is presented in [13]. They proposed a generic transformation that meets the specified requirements and that encodes a file using forward error correcting codes in order to add robustness to any Remote Data Checking (RDC) scheme based on spot checking. Our scheme has static properties and hence follows the original PDP presented in [4]. Many other variations of PDP schemes have been

proposed in the past. The techniques have been based on various concepts to verify the possession of data that a remote server stores.

The PDP solution proposed in this thesis is the first attempt to build a PDP scheme that uses the concepts of $\sum$-protocols. The security of the proof of possession of data on a server is proved using the properties of sigma protocols. Reference [4] uses various assumptions to guarantee data possession of their scheme in the random oracle model. The proposed scheme uses the discrete logarithm assumption to prove security. Reference [4] offers unlimited verifications which we achieve but we do not address public verifiability that is given in [4]. Compared to the PDP scheme in [4], the proposed scheme is more efficient in both setup and verification phases because it depends on lesser number of modular exponentiations, multiplications and additions.

2.2. Provable Data Possession (PDP)

Reference [4] gives a framework for provable data possession. A PDP protocol checks that an outsourced storage site retains a file F, which is a collection of blocks. The client C (data owner) pre-processes the file, generating a small amount of metadata that is stored locally, transmits the file along with the metadata to the server and may delete its local copy. The server stores this file. At a later time, the client issues a challenge to the server to check if the server has retained the file. The client requests that the server compute a function of the stored file and metadata, which it sends back to the client. Using its locally stored data, the client verifies the response. Several variations of this scheme have been proposed under different cryptographic assumptions. Such schemes provide probabilistic guarantees of possession, where the client checks a random subset of stored blocks with each challenge that it generates.

CHAPTER 3. BASIC CRYPTOGRAPHIC PRIMITIVES

In this chapter, the details of the cryptographic primitives that are needed to understand the main scheme are explained.

3.1. Public-key Encryption

Public-key encryption allows two parties who do not share a secret key to communicate privately. To use a public-key encryption scheme, a receiver begins by generating a public key (pk) and private key (sk). It then publishes the public key while keeping the private key secret. The sender uses the receiver's public key (pk) to encrypt messages, while the receiver uses the private key (sk) to decrypt.

3.2. Zero-knowledge Proofs

Zero-knowledge proof is an interactive proof with the additional property that the verifier learns nothing beyond the correctness of the statement being proved. In the context of cryptographic protocols, zero-knowledge proofs can be used to enforce "good behavior" by having parties prove that they indeed followed the protocol correctly. These proofs must reveal nothing about the parties' private inputs, and as such must be zero knowledge.

3.3. Proofs of Knowledge

Proof of Knowledge [14] is a proof system where the prover claims to know a certain piece of information (such as a secret key corresponding to a given public one). Such proof systems are built using knowledge completeness and knowledge soundness.

3.4. Properties of Completeness and Soundness

In any proof system, there are following properties: completeness and soundness.

3.4.1. Completeness

A correct statement can be proved within the proof system, so when the prover and verifier are honest and interact with each other on a statement that's in a language, then the verifier will accept. This property simply means that if the prover knows the claimed information and follows the protocol, he can almost always convince the verifier.

3.4.2. Soundness

You cannot prove incorrect statements in your proof system. i.e., for any x that's not in the language, the verifier will accept after interacting with any arbitrary cheating prover with at most negligible probability. Loosely speaking, this property says that if some prover can, using whatever strategy, convince the verifier with substantial probability, then the prover knows the information in question. By "knowing the information" we mean that the prover can compute it, and that the time he needs to do so is roughly inversely proportional to the probability with which the verifier gets convinced.

3.5. Sigma Protocol [15]

A proof of knowledge for a binary NP-relation, *i.e.*, R $\subseteq \{0,1\}^* \times \{0,1\}^*$, enables a prover that is given $(x, w) \in R$ to prove his knowledge of w to a verifier that knows x. The binary relation R has the restriction that if $(x, w) \in R$, then the length of w is at most $p(|x|)$ for some polynomial $p(.)$ and $|x|$ is the length of x. For some $(x, w) \in R$, we may think of *x* as an instance of some computational problem, and w as the solution to the instance. We call w a *witness* for x. We define L_R to be the set of inputs x for which there exists a w such that $(x, w) \in R$.

$\sum$-protocols ([16], [15]) are three-move protocols between a prover, P and verifier, V in which P and V have a common input x and P has a private input w such that $(x, w) \in R$. P tries to prove to V that either x belongs to language L_R or it knows a witness w such that $(x, w) \in R$. The protocol pattern is given below.

Protocol 3.1. $\sum$-protocol template for a relation R:

- Common Input: P and V get x.
- Private Input: P has a value w such that $(x, w) \in R$.
- The protocol:
    1. P sends V a message a. Let r be the randomness used to generate a.
    2. V sends a random t-bit challenge e to P.
    3. P sends a reply z, and V decides to accept or reject based on the data he has seen, i.e. (x, a, e, z).

The protocol representation is given in Figure 1.

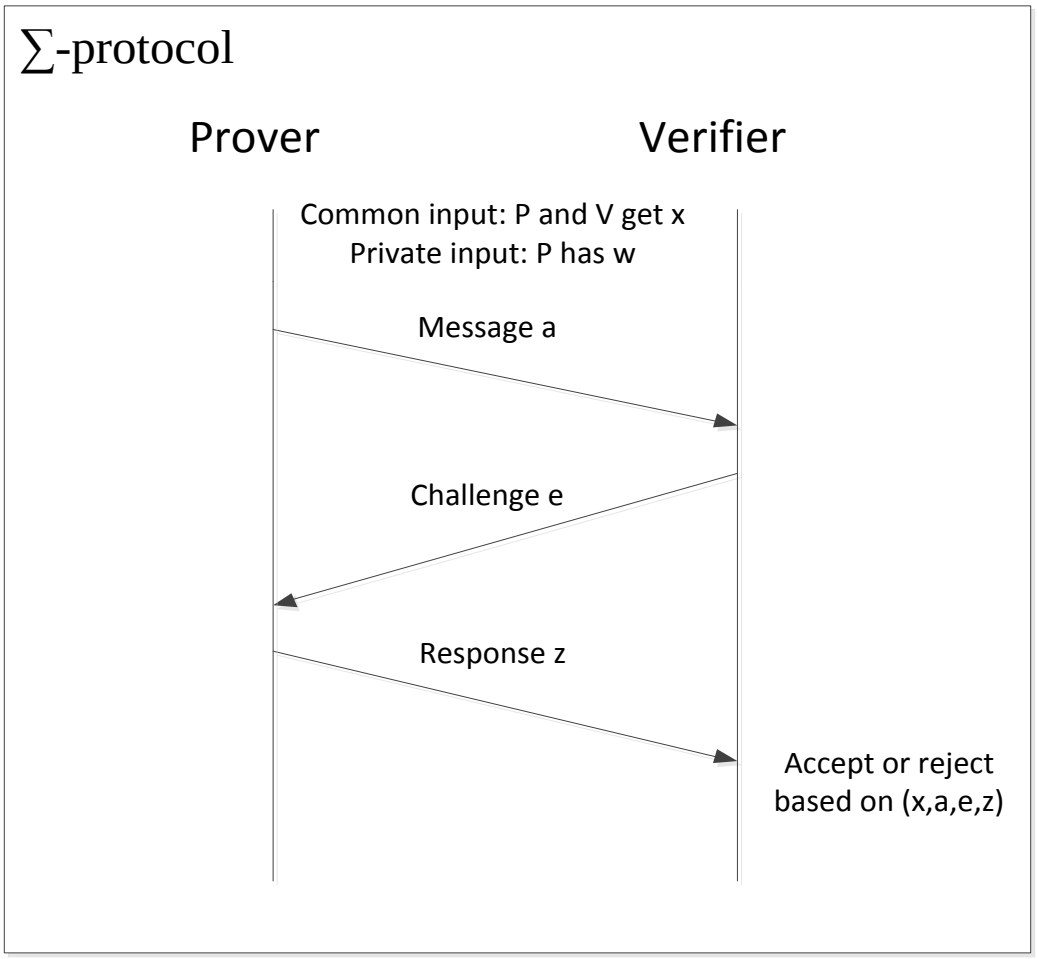

Figure 1. Protocol representation of ∑-protocol template

We will assume throughout that both *P*, *V* are probabilistic polynomial time machines, so *P*'s only advantage over *V* is that he knows w. If *V* accepts then it is convinced that *P* knows a witness w such that $(x, w) \in R$. Moreover, if *V* is honest it does not learn any more information than $(x, w) \in R$. This implies that the ∑-protocol is honest-verifier zero-knowledge. Note that the relation R is such that it is hard to compute w from x, otherwise *V* could compute w by itself and it does not need to be convinced that *P* knows w.

Like any proof system a ∑-protocol is only useful if it satisfies a notion of completeness and soundness. Completeness guarantees that if a honest prover (i.e., one that follows the protocol) indeed "knows" a witness w such that $(x, w) \in R$, then an honest verifier will be convinced. Soundness, on the other hand, guarantees that if the verifier accepts an interaction with a prover, then the prover indeed "knows" a witness w such that $(x, w) \in R$. The soundness of a sigma-protocol can be defined in a variety of ways (see [10] for a discussion), but is typically formalized using the notion of special soundness [17]. Informally, special soundness requires that one be able to efficiently "extract" a witness from the transcripts of two instances of the protocol that use the same commitment but different challenges.

We now define ∑-protocols formally.

Definition 3.1 [15]**.** A protocol *P* is said to be a ∑-protocol for relation *R* if:

- *P* is of the above 3-move form, and we have completeness: if *P*, *V* follow the protocol on the input x and private input w to *P* where $(x, w) \in R$, the verifier always accepts.
- From any *x* and any pair of accepting conversations on input $x, (a, e, z), (a, e', z')$ where $e \neq e'$, one can efficiently compute w such that $(x, w) \in R$. This is called the soundness property.
- Define L_R to be the set of x's for which there exists w such that $(x, w) \in R$. There exists a polynomial-time simulator M, which on input $x \in L_R$ and a random e outputs an accepting conversation of the form (a, e, z), with the same probability distribution as conversations between the honest *P*, *V* on input x. This is called special honest-verifier zero-knowledge.

Reference [16] shows that a sigma-protocol can be converted to zero-knowledge proofs or ZKPOK using commitment schemes. In this paper, we use ∑-protocols that are witness indistinguishable and witness hiding to construct an efficient one-time signature scheme [15] given in Section 3.7.

3.6. Okamoto Protocol

We consider an example one-time signature scheme that is based on the Okamoto protocol, an example ∑-protocol. Let G_q be a group of prime order q, with generators g_1 and g_2, set in such a way that no one can efficiently compute, x, such that $g_1 = g_2^x$. The Okamoto protocol is a ∑-protocol based on the relation $R = \{(h, (w_1, w_2)) | \ h = g_1^{w_1} g_2^{w_2}\}$. *P* and *V* have a common input $h = {g_1}^{w_1} {g_2}^{w_2}$. *P* has private input (w_1, w_2) such that $(h, (w_1, w_2)) \in R$. The Okamoto protocol proceeds as follows:

Protocol 3.2. The Okamoto Protocol:

- Common Input: *P* and *V* get $h = g_1^{w_1} g_2^{w_2}$.
- Private Input: *P* chooses a value (w_1, w_2) such that $(h, (w_1, w_2)) \in R$.
- The protocol:
    1. *P* sends *V* a message $a = g_1^{v_1} g_2^{v_2}$, where (v_1, v_2) is chosen uniformly at random from Z_q.
    2. *V* sends *P* a random challenge e in Z_q.
    3. *P* computes $z_1 = ew_1 + v_1 \ mod \ q$, $z_2 = ew_2 + v_2 \ mod \ q$ and sends $z = (z_1, z_2)$ to *V*. *V* accepts if $g_1^{z_1} g_2^{z_2} = ah^e$.

The protocol representation is given in Figure 2.

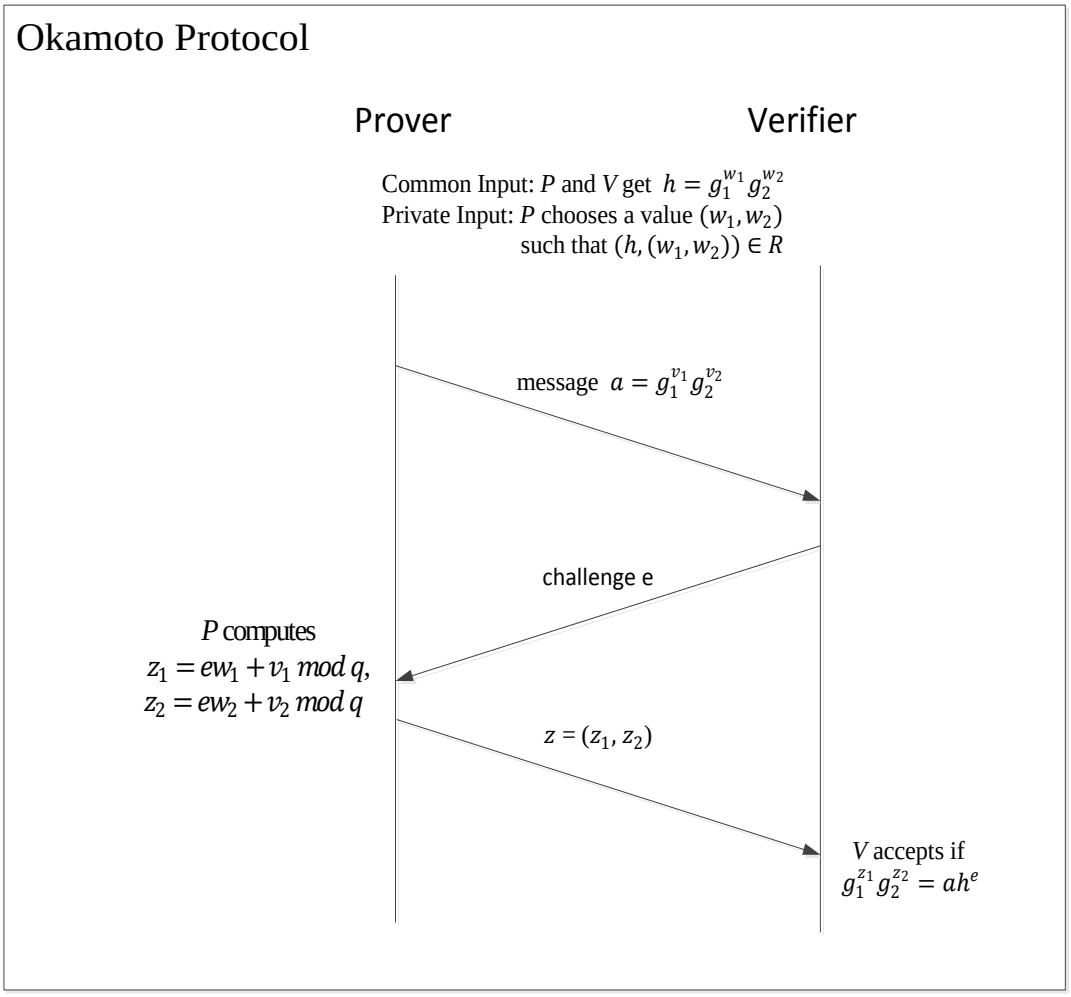

Figure 2. Protocol representation of Okamoto-protocol

3.6.1. Properties of witness-indistinguishability and witness-hiding

The properties of witness-indistinguishability and witness-hiding also hold for Okamoto protocols. They are defined as follows:

Definition 3.2 [17]**.** Perfect witness-indistinguishable (WI): No matter how e is chosen by the verifier, all possible pairs (w_1, w_2) consistent with h(i.e., satisfying $g_1^{z_1} g_2^{z_2} = ah^e$) are equally likely from the point of view of the verifier.

Definition 3.3 [17]. Witness hiding (WH): Even a malicious verifier cannot, by talking to the prover, learn the underlying witnesses.

3.7. One-time Signature Scheme

We now build a one-time signature scheme based on the Okamoto protocol [15]. It consists of three probabilistic polynomial time algorithms $(Gen, Sign, Verify)$.

One-time signature scheme based on the Okamoto protocol:

- Gen: The verification key (pk) and signing key (sk) are: $pk = (h, a) = (g_1^{w_1} g_2^{w_2} mod\ p,\ g_1^{v_1} g_2^{v_2} mod\ p), sk = ((w_1, w_2), (v_1, v_2))$.
- $Sign$: The signature on a message m is computed as $\sigma = (z_1, z_2) = (mw_1 + v_1\ mod\ q, mw_2 + v_2\ mod\ q)$.
- $Verify$: The receiver verifies a message-signature pair (m, σ) by checking if the following equality holds: $g_1^{z_1} g_2^{z_2} = ah^m$.

Note that $g_1^{z_1} g_2^{z_2} = g_1^{mw_1+v_1}\ g_2^{mw_2+v_2} =\ g_1^{v_1} g_2^{v_2}\ (g_1^{w_1} g_2^{w_2})^m = ah^m$.

This one-time signature scheme is secure because an adversary having seen at most one valid signature cannot efficiently compute a valid signature on a different message. Suppose that there exists an adversary that can compute a valid signature on a different message, given one valid signature, then the adversary has two different signatures σ and σ' for the same $pk\ = (h, a)$. This implies that the adversary has two different conversations for the underlying $\sum$-protocol (h, a, m, σ) and (h, a, m', σ') $(note\ m\ \neq\ m')$. Therefore, from the special soundness property of the protocol the adversary can compute (w_1, w_2). However this contradicts the witness hiding property of $\sum$-protocols. Thus, there exists no adversary that can efficiently

compute a valid signature on a different message, given one valid signature. The above one-time signature scheme is therefore secure.

Since the above signature is a one-time signature it can only be used to sign one message with one key (sk, pk). Every new message must be signed with a new key to maintain security.

We propose to change the key by simply changing (v_1, v_2) while keeping (w_1, w_2) unchanged. The signing key (or private key of the signer) for signing the i^{th} message m_i is $sk = ((w_1, w_2), (v_{1i}, v_{2i}))$, where (w_1, w_2) is such that $(h, (w_1, w_2)) \in R$, for $h = g_1^{w_1} g_2^{w_2}$, and (v_{1i}, v_{2i}) are chosen at random from Z_q for each m_i. The verification key (or public key) is $pk_i = (g_1^{w_1} g_2^{w_2}, g_1^{v_{1i}} g_2^{v_{2i}}) = (h, a_i)$. The verification key has to be distributed to each receiver of message m_i. We assume that this can be accomplished using authenticated message transmission. The signer of the message block can select n pairs of random values $((v_{11}, v_{21}), (v_{12}, v_{22}), \dots, (v_{1n}, v_{2n}))$ from Z_q and compute the n verification keys, $pk_1, pk_2, \dots, pk_n$. Note that $h = g_1^{w_1} g_2^{w_2}$, g_1 and g_2 are known to the receiver in the beginning. The signer computes the signature of the i^{th} message m_i as $\sigma_i = (z_{1i}, z_{2i}) = (m_i w_1 + v_{1i} \bmod q, m_i w_2 + v_{2i} \bmod q)$ and sends the message-signature pair (m_i, σ_i) to the receiver. Upon receipt of a message (m_i, σ_i), the receiver performs verification by checking if the following equality is satisfied: $g_1^{z_{1i}} g_2^{z_{2i}} = a_i h^{m_i}$. In order to verify the message-signature pair, $(m_i, \sigma_i = (z_{1i}, z_{2i}))$, the receiver should know the values g_1, g_2, h, and a_i. The receiver receives g_1, g_2 and h in the beginning. The sender computes n verification keys and sends them to all receivers so the receiver knows a_i.

3.8. Pseudorandom Function (PRF)

A pseudorandom function [18] is a function that cannot be distinguished from a truly random function. A keyed function *F* is a two-input function F: $\{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^*$, where the first input is called the key and denoted by k, and the second input is just called the input. F_k: $\{0,1\}^* \rightarrow \{0,1\}^*$ defined by $F_k(x) \stackrel{\text{def}}{=} F(k,x)$. On fixing a key $k \in \{0,1\}^n$ we obtain a function $F_k(.)$ mapping n-bit strings to n-bit strings. *F* is efficient if there is a deterministic polynomial-time algorithm that computes $F(k,x)$ given k and x as input. *F* is pseudorandom if the function F_k(for randomly chosen k) is indistinguishable from a function chosen uniformly at random from the set of all functions having the same domain and range, that is, if no polynomial-time adversary can distinguish whether it is interacting with F_k or *f* (where *f* is chosen at random from the set of all functions mapping n-bit strings to n-bit strings). From theoretical point of view, it is known that pseudorandom function exist if and only if pseudorandom generators exist, and so pseudorandom functions can be constructed based on any of the hard problems that allow the construction of pseudorandom generators. PRFs were introduced by Goldreich, Goldwasser and Micali [19] and are known to exist under the assumption that one-way functions exist [20].

3.9. Pseudorandom Permutation (PRP)

There exists some keyed permutation *F* [18] for which it is impossible to distinguish in polynomial time between interactions with F_k(for randomly-chosen key k) and interactions with a truly random permutation. Let F: $\{0,1\}^* \times \{0,1\}^* \rightarrow \{0,1\}^*$ be an efficient, length-preserving, keyed function. We call *F* a keyed permutation if for every k, the function $F_k(.)$ is one-to-one. We say that a keyed permutation is efficient if there is a polynomial-time algorithm computing $F_k(x)$ given k and x, as well as a polynomial-time algorithm computing $F_k^{-1}(x)$ given k and x.

The only change is that we now require that F_k(for randomly chosen k) be indistinguishable from a randomly-chosen permutation rather than a randomly chosen function.

3.10. Pseudorandom-Generator

As given by Ateniese $et\ al.$ [4], we make use of π, a pseudo-random permutation (PRP) and *f* a pseudo-random function (PRF) that takes as input a security parameter k with the following parameters:

$$\pi: \{0,1\}^k \times \{0,1\}^{log_2(n)} \rightarrow \{0,1\}^{log_2(n)};$$

$$f: \{0,1\}^k \times \{0,1\}^{log_2(n)} \rightarrow \{0,1\}^l$$

We write $f_k(x)$ to denote *f* keyed with key k applied on input x. The purpose of including the coefficients b_j in the values for σ computed by the server as in [4] is to ensure that the server possesses each one of the requested blocks (see Section 4.2 for details). These coefficients are determined by a PRF keyed with a fresh randomly-chosen key for each challenge to prevent the server from storing combinations (e.g., sums) of the original blocks instead of the original file blocks themselves.

CHAPTER 4. ∑-PROVABLE DATA POSSESSION

The client C wants to store a file F, a finite ordered collection of n blocks: F= $(m_1, m_2, \ldots, m_n)$ on the server S. We denote the output y of an algorithm B by $y \leftarrow B$.

We first give the details of the algorithms used in the main scheme and then define a ∑-provable data possession scheme that follows [4]. We later present a security definition that demonstrates the data possession property.

4.1. Details of the Proposed ∑-PDP Scheme

C chooses a value (w_1, w_2) such that $(h, (w_1, w_2)) \in R$. (w_1, w_2) corresponds to the witness in the Okamoto protocol and is the client's secret chosen before the execution of the protocol. We propose the witness to remain unchanged throughout. Let G_q be a group of prime order q, with generators g_1 and g_2, set in such a way that no one can efficiently compute x, such that $g_1 = g_2^x$. Instead of choosing the secret parameters $((v_{11}, v_{21}), (v_{12}, v_{22}), \ldots, (v_{1n}, v_{2n}))$ at random we use a pseudorandom function to generate them to sign the message blocks and later to verify the proof of possession. The client deletes these secret parameters from its storage and can generate them when needed. For $1 \leq i \leq n$, C generates $((v_{11}, v_{21}), (v_{12}, v_{22}), \ldots, (v_{1n}, v_{2n}))$ from a pseudorandom function $f_{k_v}(i)$ and $f_{k_v}(i+1)$. Given a secret key k_v, $v_{1i} = f_{k_v}(i), v_{2i} = f_{k_v}(i+1)$.

As defined earlier, let *f* be a pseudo-random function and π be a pseudo-random permutation.

A detailed description of the algorithms used in building the ∑- PDP scheme based on the Okamoto protocol are as follows:

4.1.1. $KeyGen(1^k)$

$KeyGen(1^k)$ is a probabilistic polynomial algorithm that generates the public key pk_i and secret key sk pairs (pk_i, sk) such that $pk_i = (h, a_i) = (g_1^{w_1} g_2^{w_2}, g_1^{v_{1i}} g_2^{v_{2i}})$ for each m_i for all $1 \leq i \leq n$ and $sk = ((w_1, w_2), ((v_{11}, v_{21}), (v_{12}, v_{22}), \dots, (v_{1n}, v_{2n})))$. It takes a security parameter k, generators g_1 and g_2, witness (w_1, w_2) as input, and returns a pair of public and secret key (pk_i, sk). The public key (also the verification key) is $pk_i = (g_1^{w_1} g_2^{w_2}, g_1^{v_{1i}} g_2^{v_{2i}}) = (h, a_i)$. The public key on each message m_i is given to the server. This is done at the beginning of the protocol. Private Input: For $1 \leq i \leq n$, C generates $((v_{11}, v_{21}), (v_{12}, v_{22}), \dots, (v_{1n}, v_{2n}))$ in the following manner:

$$(v_{1i}, v_{2i}) = (f_{k_v}(i), f_{k_v}(i+1))$$

and sets the secret key $sk = ((w_1, w_2), ((v_{11}, v_{21}), (v_{12}, v_{22}), \dots, (v_{1n}, v_{2n})))$. Common Input: C gives S public key $pk_i = (h, a_i) = (g_1^{w_1} g_2^{w_2}, g_1^{v_{1i}} g_2^{v_{2i}})$ for each m_i for all $1 \leq i \leq n$. These public keys are used as verification keys to check the validity of the signatures that the server receives from the client on the corresponding messages.

4.1.2. $Sign(sk, m_i)$

It computes signature σ_{i,m_i} on the file block m_i for all $1 \leq i \leq n$. The signatures are computed in the following manner:

$\sigma_{i,m_i} = (m_i w_1 + v_{1i} \, mod \, q, m_n \, w_2 + v_{2i} \, mod \, q) = (z_{1i}, \, z_{2i})$.

It returns $F = (m_1, m_2, \dots, m_n)$ and $\sum = (\sigma_{1,m_1}, \sigma_{2,m_2}, \dots, \sigma_{n,m_n})$.

4.1.3. $Ver(pk_i, m_i, \sigma_{i,m_i})$

$Ver(pk_i, m_i, \sigma_{i,m_i})$ is an algorithm that checks the validity of the signature σ_{i,m_i} corresponding to message m_i that it received from the client for $1 \leq i \leq n$. It takes a public key pk_i, signature σ_{i,m_i} and message block m_i and checks if σ_{i,m_i} is a valid signature on m_i. The client computes the signature of the i^{th} message m_i as $\sigma_{i,m_i} = (z_{1i}, z_{2i}) = (m_i w_1 + v_{1i} \ mod \ q, m_i w_2 + v_{2i} \ mod \ q)$ and sends (m_i, σ_{i,m_i}) to all receivers. On receiving (m_i, σ_{i,m_i}), the server performs verification by checking if the following equality is satisfied: $g_1^{z_{1i}} g_2^{z_{2i}} = a_i h^{m_i}$. The server receives g_1, g_2, and h in the beginning. To verify the message-signature pair, $(m_i, \sigma_{i,m_i} = (z_{1i}, z_{2i}))$, the server should know the values g_1, g_2, h, and a_i. C computes n public keys (also verification keys) and sends them to S so the server knows a_i in the beginning. It returns "$success$" if σ_{i,m_i} is a correct signature on m_i. Otherwise outputs "$failure$".

4.1.4. $GenProof(F = (m_1, m_2, \ldots, m_n), e, \sum = (\sigma_{1,m_1}, \sigma_{2,m_2}, \ldots, \sigma_{n,m_n}))$

1. A pair of keys (k_1, k_2) is chosen by C. The challenge e is chosen in the following manner. For $1 \leq j \leq c$, it computes the indices, i_j of the blocks and co- efficients, b_j for which the proof is generated.

$$i_j = \pi_{k_1}(j)$$

$$b_j = f_{k_2}(j)$$

Therefore,

Challenge $e = ((i_1, i_2, \ldots, i_c), (b_1, b_2, \ldots, b_c))$ is chosen where $1 \leq c \leq n$.

2. Compute message $M = (b_1 m_{i_1} + b_2 m_{i_2} + \ldots + b_c m_{i_c}) mod\ q$ and $(s_1, s_2) =$ $(b_1 \sigma_{i_1, m_{i_1}} + b_2 \sigma_{i_2, m_{i_2}} + \ldots . + b_c \sigma_{i_{c, m_{i_c}}})\ mod\ q =$ $(b_1((z_{11}, z_{21})) + b_2((z_{12}, z_{22})) + \ldots . + b_c((z_{1c}, z_{2c})))mod\ q = ((b_1 z_{11} + b_2 z_{12} + \ldots . + b_c z_{1c}), (b_1 z_{21} + b_2 z_{22} + \ldots . + b_c z_{2c})$. Note: ($\sigma_{i_j, m_{i_j}}$ is the i_j-th value in $\sum$).
3. Output proof $(M, (s_1, s_2))$.

4.1.5. $\boldsymbol{CheckProof(pk, sk, e, (M, (s_1, s_2)))}$

1. For $1 \leq j \leq c$, C generates $(v_{11}, v_{21}), (v_{12}, v_{22}), \ldots, (v_{1c}, v_{2c})$ in the following manner:

$$(v_{1i}, v_{2i}) = (f_{k_v}(i), f_{k_v}(i + 1))$$

2. Compute

$V \leftarrow g_1^{b_1 v_{11}}\ g_1^{b_2 v_{12}} \ldots\ g_1^{b v_{1c}}\ \ g_2^{b_1 v_{21}}\ g_2^{b_2 v_{22}} \ldots\ g_2^{b_c v_{2c}} mod\ p =$ $g_1^{(b_1 v_{11} + b_2 v_{12} + \ldots . + b_c v_{1c})}\ g_2^{(b_1 v_{21} + b_2 v_{22} + \ldots . + b_c v_{2c})}\ mod\ p$. Set private key $pk \leftarrow$ $(h, V) = ((g_1^{w_1} g_2^{w_2}), (g_1^{(b_1 v_{11} + b_2 v_{12} + \ldots + b_c v_{1c})}\ g_2^{(b_1 v_{21} + b_2 v_{22} + \ldots + b_c v_{2c})})$ and secret key $sk \leftarrow ((w_1, w_2), ((v_{11}, v_{21}), (v_{12}, v_{22}), \ldots, (v_{1c}, v_{2c})))$. For $1 \leq\ j \leq c$, compute the indices, i_j of the blocks and co- efficients, b_j for which the proof is generated.

$$i_j = \pi_{k_1}(j)$$

$$b_j = f_{k_2}(j)$$

3. Given $(pk, sk, (M, (s_1, s_2))$ and $e = ((i_1, i_2, \ldots, i_c), (b_1, b_2, \ldots, b_c))$ (for $1 \leq j\ \leq\ c$), if $g_1^{s_1} g_2^{s_2} = Vh^M$ output "$success$". Otherwise output "$failure$". Note that $g_1^{s_1} g_2^{s_2} =$

$$g_1^{((b_1 m_1 + b_2 m_2 + \ldots + b_c m_c) w_1 + (b_1 v_{11} + b_2 v_{12} + \ldots + b_c v_{1c}))} *$$

$$g_2^{((b_1 m_1 + b_2 m_2 + \ldots + b_c m_c) w_2 + (b_1 v_{21} + b_2 v_{22} + \ldots + b_c v_{2c}))} mod\ p =$$

$$g_1^{(b_1 v_{11} + b_2 v_{12} + \ldots + b_c v_{1c})} * g_2^{(b_1 v_{21} + b_2 v_{22} + \ldots + b_c v_{2c})} *$$

$$(g_1^{w_1} g_2^{w_2})^{(b_1 m_1 + b_2 m_2 + \ldots + b_c m_c)} mod\ p\ = Vh^M.$$

4.2. ∑-Provable Data Possession

Definition 5.1. (∑-Provable Data Possession (∑-PDP)) [21]. A ∑-PDP scheme is a tuple of five polynomial-time algorithms $(KeyGen, Sign, Ver, GenProof, CheckProof)$ such that:

- $KeyGen(1^k) \rightarrow (pk_i, sk)$ is a probabilistic key generation algorithm that is run by the client to generate the public and private key to the scheme. It takes a security parameter k as input, and returns a pair of public and secret keys (pk_i, sk) for each file block m_i.
- $Sign(sk, m_i) \rightarrow \sigma_{m_i}$ is an algorithm run by the client to generate the signature for each message block for $1 \leq i \leq n$. It takes as inputs a secret key sk and a file block m_i, and returns the signature σ_{i,m_i}.
- $Ver(pk_i, m_i, \sigma_{i,m_i}) \rightarrow \{"success", "failure"\}$ is run by the server to check the validity of the signature σ_{i,m_i} corresponding to message m_i that it received from the server for $1 \leq i \leq n$. It takes a public key pk_i, signature σ_{i,m_i} and message block m_i and returns "$success$" if σ_{i,m_i} is a correct signature. Otherwise outputs "$failure$".
- $GenProof(F, e, \sum) \rightarrow (M, (s_1, s_2))$is an algorithm run by the server to generate a proof of possession. It takes as inputs an ordered collection of F, a challenge e from the client C and ∑ which is an ordered collection of signatures corresponding to the

blocks in F sent by C. It returns a proof of possession $(M, (s_1, s_2))$ for the blocks in F that are determined by the challenge e.

- $CheckProof(pk, sk, e, (M, (s_1, s_2)) \rightarrow \{\text{“}success\text{”}, \text{“}failure\text{”}\}$ is run by the client in order to validate a proof of possession. It takes as inputs a public key pk, a secret key sk, a challenge e and proof of possession $(M, (s_1, s_2))$. It outputs whether $(M, (s_1, s_2))$ is a correct proof of possession for the blocks given by e.

4.3. ∑-PDP Scheme

We now construct a ∑-PDP scheme in two phases, *Setup* and *Challenge*:

Setup: The client C runs $(pk_i, sk) \leftarrow KeyGen(1^k)$, gives pk to S and keeps sk secret. C then runs $\sigma_{i,m_i} \leftarrow Sign(sk, m_i)$ for all $1 \leq i \leq n$ and sends pk, $F = m_1, m_2, \ldots, m_n$ and $\sum = (\sigma_{1,m_1}, \sigma_{2,m_2}, \ldots, \sigma_{n,m_n})$ to S for storage. S may run $\{\text{“}success\text{”}, \text{“}failure\text{”}\} \leftarrow Ver(pk_i, m_i, \sigma_{i,m_n})$ to check the validity of the signatures it received for $1 \leq i \leq n$. C may now delete F and $\sum$ from its local storage.

Challenge: C requests proof of possession for c distinct blocks of file F (with $1 \leq c \leq n$):

1. C generates a random challenge e and sends it to S.
2. S runs $(M, (s_1, s_2)) \leftarrow GenProof(F, e, \sum)$ and sends $(M, (s_1, s_2))$ to C.
3. C runs $CheckProof(pk, sk, e, (M, (s_1, s_2)))$ to check if $g_1^{s_1} g_2^{s_2} = Vh^M$.

In the *Setup* phase, C generates signatures for each file block and stores them along with the file at S. In the *Challenge* phase, C requests proof of possession for a challenge which is a subset of the blocks in F.

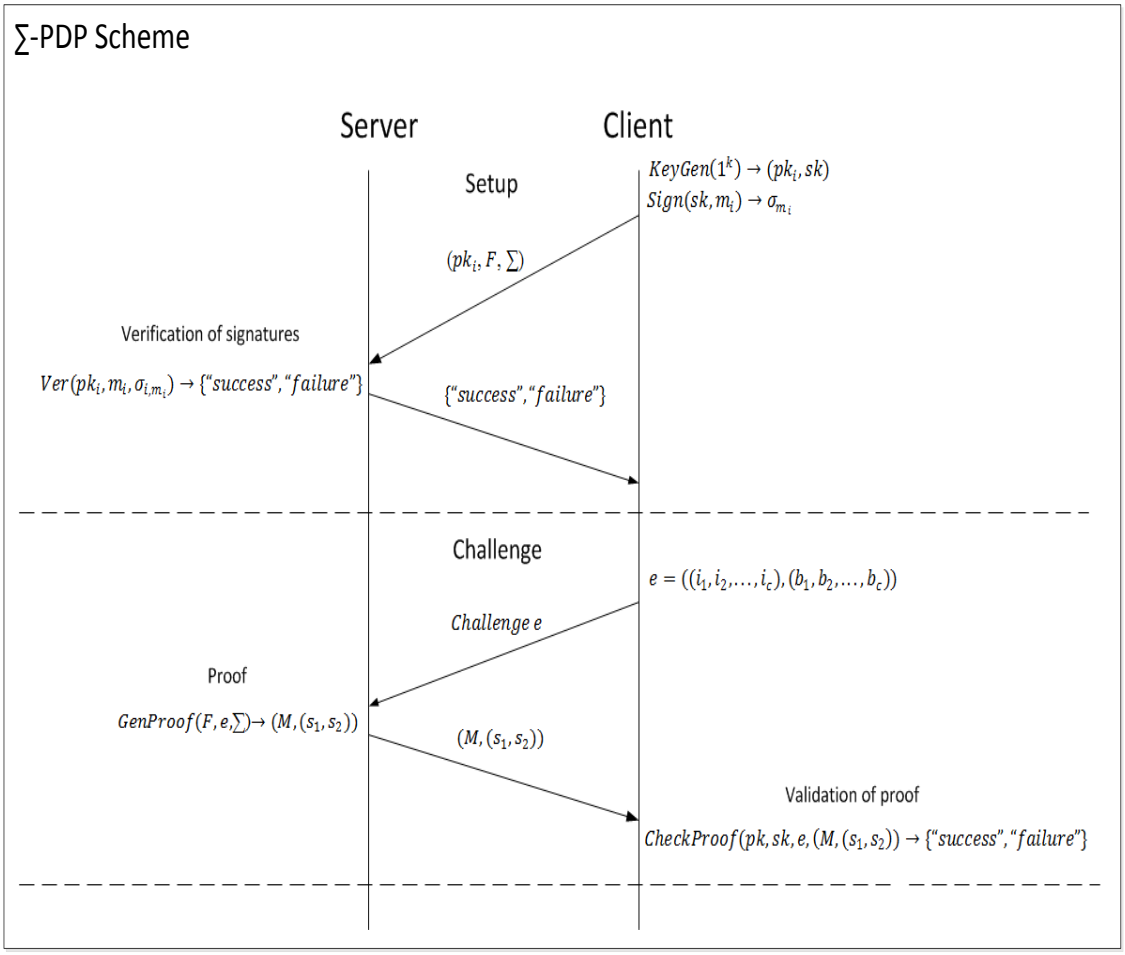

Figure 3. ∑-PDP scheme

We use the data possession game (defined below) [4] to show security of the ∑-PDP scheme. The Data Possession Game shows that an adversary cannot successfully construct a valid proof without possessing all blocks corresponding to a given challenge, unless it guesses all the missing blocks correctly.

4.4. Data Possession Game

- ***Setup**:* The challenger runs algorithm $KeyGen(1^k)$, to obtain pk_i and sk. It sends pk_i to the adversary and keeps sk secret.

- ***Query:*** The adversary adaptively selects some file block m_1 and sends it to the challenger. The challenger computes the verification signature $\sigma_{1,m_1} \leftarrow Sign(sk, m_1)$ and sends it back to the adversary. The adversary continues to query the challenger for the verification signatures $\sigma_{2,m_2}, \sigma_{3,m_3}, \ldots, \sigma_{n,m_n}$ on the blocks of its choice $m_2, m_3, \ldots, m_n$. The challenger generates σ_{i,m_i} for $1 \leq i \leq n$, by computing $\sigma_{i,m_i} \leftarrow Sign(pk_i, sk, m_i)$. The adversary stores all the blocks as an ordered collection $F = (m_1, m_2, \ldots, m_n)$ along with the corresponding verification signatures $\sum = (\sigma_{1,m_1}, \sigma_{2,m_2}, \ldots, \sigma_{n,m_n})$.
- ***Challenge****:* The challenger generates a challenge e for the file blocks $m_{i_1}, m_{i_2}, \ldots, m_{i_c}$ where $1 \leq i_j \leq n, 1 \leq j \leq c, 1 \leq c \leq n$ and requests the adversary to provide a proof of possession for these blocks determined by e.
- ***Forge****:* The adversary computes a pair of values as proof of possession $(M, (s_1, s_2))$ for the blocks indicated by e and returns $(M, (s_1, s_2))$.

If $CheckProof\ (pk, sk, e, (M, (s_1, s_2))) =$ "$success$", then the adversary has won the Data Possession Game.

Definition 4.2 [4]**.** A $\sum$-PDP system (*Setup*, *Challenge*) built on a PDP scheme $(KeyGen, Sign, GenProof, CheckProof)$ guarantees data possession if for any (probabilistic polynomial-time) adversary *A* the probability that *A* wins the Data Possession Game on the set of file blocks is negligibly close to the probability that the challenger can extract those file blocks by means of a knowledge extractor *K*.

As given by Ateniese *et al.* [4], in the security definition, the notion of a knowledge extractor is similar to the one introduced in the context of proof of knowledge [22]. If the

adversary is able to win the Data Possession Game, then *K* can execute $GenProof$ repeatedly until it extracts the selected blocks. On the other hand, if *K* cannot extract the blocks, then the adversary cannot win the game with more than negligible probability.

4.5. Proof of Security

Theorem 4.3. If there exists a hard discrete logarithm problem that cannot be computed efficiently then there exists a ∑-PDP that guarantees data possession.

Proof: Under the Discrete Logarithm (DL) assumption of Okamoto protocols, we prove that ∑-PDP guarantees data possession.

We assume that a group G_q with prime order q is known. Furthermore, two generators g_1 and g_2 of G_q are known. These values have been set up in such a way that nobody knows $log_{g_2} g_1$, i.e., we assume that no one can efficiently compute x such that $g_1 = g_2^x$. This is the DL assumption.

We assume there exists an adversary $\mathcal{B}$ that wins the Data Possession Game on a challenge picked by $\mathcal{A}$ and show that $\mathcal{A}$ will be able to extract the file blocks determined by the challenge. If $\mathcal{B}$ wins the data possession game of the ∑-PDP scheme by sending a valid proof that contains a message that is not same as the correct message, then we show how to construct an adversary $\mathcal{A}$ that uses $\mathcal{B}$ in order to find x such that $g_1 = g_2^x$. This violates the DL assumption. Therefore, $\mathcal{B}$ must generate a proof that contains a message identical to the correct message. A knowledge extractor *K* is then constructed which can extract the file blocks involved in the proof. $\mathcal{A}$ will play the role of the challenger in the Data Possession Game and will interact with $\mathcal{B}$.

$\mathcal{A}$ simulates a $\sum$-PDP environment for $\mathcal{B}$ as follows:

Setup:

$\mathcal{A}$ chooses $g_1, g_2 \in G_q$. The secret key values (w_1, w_2) are chosen at random from the group $Z_q$and $((v_{11}, v_{21}), (v_{12}, v_{22}), \dots, (v_{1n}, v_{2n}))$ are generated by a PRF with a secret key k_v. The witnesses (w_1, w_2) remain constant throughout the execution of the protocol and are hence chosen before the start of the protocol. The parameters $((v_{11}, v_{21}), (v_{12}, v_{22}), \dots, (v_{1n}, v_{2n}))$ are chosen at the beginning of the protocol.

$\mathcal{A}$ computes ${g_1}^{w_1} {g_2}^{w_2} = h$ and $g_1^{v_{1i}} g_2^{v_{2i}} = a_i$ and sets public key $pk_i \leftarrow (h, a_i) = (g_1^{w_1} g_2^{w_2},\ g_1^{v_{1i}} g_2^{v_{2i}})$ on the file block m_i for all $1 \leq\ i \leq\ n$; secret key $sk \leftarrow ((w_1, w_2), ((v_{11}, v_{21}), (v_{12}, v_{22}), \dots, (v_{1n}, v_{2n})))$.

It sends pk_i to $\mathcal{B}$ and keeps sk secret. (see Section IV-D for details on checking the validity of the signatures at the server on knowing the public key)

Query:

$\mathcal{B}$ makes signing queries adaptively: $\mathcal{B}$ selects a block m_1 and index i_1. $\mathcal{B}$ sends m_1 and i_1 to $\mathcal{A}$, $\mathcal{A}$ generates σ_{i_1,m_1} at random and sends it back to $\mathcal{B}$. $\mathcal{B}$ continues to query $\mathcal{A}$ for the signatures $\sigma_{i_2,m_2}, \dots, \sigma_{i_n,m_n}$ on the blocks $m_2, \dots, m_n$. The restriction is that $\mathcal{B}$ cannot make tagging queries for two different blocks using the same index. $\mathcal{A}$ answers $\mathcal{B}$'s signing oracle queries as follows:

—when $\mathcal{A}$ receives a signing query for a block m and index i, with $1 \leq i \leq n$:

if a previous tagging query has been made for the same m and i, then $\mathcal{A}$ retrieves the recorded tuple $(m, i, (r_{1i}, r_{2i}))$ and returns $\sigma_{i,m} = (r_{1i}, r_{2i})$.

else, $\mathcal{A}$ picks $(r_{1i}, r_{2i}) \overset{R}{\leftarrow} Z_q$, records the tuple $(m, i, (r_{1i}, r_{2i}))$and returns the signature $\sigma_{i,m} = (r_{1i}, r_{2i})$.

Challenge:

$\mathcal{A}$ generates the challenge $e = ((i_1, i_2, \ldots, i_c), (b_1, b_2, \ldots, b_c))$, where $i_1, i_2, \ldots, i_c$ are the indices of the blocks and $(b_1, b_2, \ldots, b_c)$ are corresponding co-efficients of the blocks for which $\mathcal{A}$ requests proof of possession (with $1 \leq i_j \leq n, 1 \leq j \leq c, \ 1 \leq c \leq n$ and $b_j$is chosen at random from Z_q). $\mathcal{A}$ sends e to $\mathcal{B}$ and requests a proof of possession for this challenge.

Forge:

$\mathcal{B}$ generates a proof $(M, (s_1, s_2))$ on the blocks $m_{i_1}, m_{i_2}, \ldots, m_{i_c}$. Note that $(M, (s_1, s_2))$ is a valid proof if it passes $CheckProof(pk, sk, e, (M, (s_1, s_2))$. $\mathcal{B}$ returns $(M, (s_1, s_2))$ to $\mathcal{A}$ and $\mathcal{A}$ checks the validity of this proof.

Let the correct proof be $(M'(s_1', s_2'))$ where $M' = b_1 m_{i_1} + b_2 m_{i_2} + \ldots + b_c m_{i_c}$. $\mathcal{A}$ knows this because it knows pk_i and sk.

If $\mathcal{B}$ sent a valid proof such that $M' \neq M$, then $\mathcal{A}$ knows two sets of values that satisfy the verifiability condition:

$$g_1^{s_1} . g_2^{s_2} = h^M V \qquad \ldots\ldots\ldots\ldots\ldots\ldots (1)$$

$$g_1^{s_1'} . g_2^{s_2'} = h^{M'} V \qquad \ldots\ldots\ldots\ldots\ldots\ldots (2)$$

On dividing (1) by (2),

$$g_1^{s_1-s_1'}.g_2^{s_2-s_2'} = h^{M-M'} = g_1^{w_1(M-M')}g_2^{w_2(M-M')}$$

(Note that $h = g_1^{w_1}g_2^{w_2}$)

This implies

$$g_1^{(s_1-s_1')-w_1(M-M')} = g_2^{(s_2-s_2')-w_2(M-M')}$$

$$g_1 = g_2^{(s_2-s_2')-w_2(M-M').(s_1-s_1')-w_1(M-M')^{-1}}$$

Therefore, an x such that $g_1 = g_2^x$ can be computed using the above as follows:

$$x = (s_2 - s_2') - w_2(M - M').(s_1 - s_1') - w_1(M - M')^{-1}$$

This violates the DL-assumption. Therefore, $\mathcal{B}$'s proof is such that $M' = M$. Hence $\mathcal{A}$ is successfully able to extract the correct message M'.

At the end of the simulation $\mathcal{A}$ will be able to extract $M' = b_1 m_{i_1} + b_2 m_{i_2} + \ldots + b_c m_{i_c}$. We now show that our protocol constitutes a proof of knowledge of the blocks m_{i_1}, $m_{i_2}, \ldots, m_{i_c}$ when $b_1, b_2, \ldots, b_c$ are pairwise distinct. We show that a knowledge extractor *K* may extract the file blocks m_{i_1}, $m_{i_2}, \ldots, m_{i_c}$. Note that each time *K* runs the $\sum$-PDP protocol, he obtains a linear equation of the form $M' = b_1 m_{i_1} + b_2 m_{i_2} + \ldots + b_c m_{i_c}$. By choosing independent coefficients $b_1, b_2, \ldots, b_c$ in c executions of the protocol on the same blocks $m_1, m_2, \ldots, m_c$, *K* obtains c independent linear equations in the variables m_{i_1}, $m_{i_2}, \ldots, m_{i_c}$, *K* may now solve these equations to obtain the file blocks m_{i_1}, $m_{i_2}, \ldots, m_{i_c}$. This shows that our proof is a proof of knowledge.

4.6. Complexity Analysis and Comparison

We follow the specifications for a scalable solution given by Ateniese *et al*. [4] where the block access and hence the computation at the server should be minimized, because the server may be involved in concurrent interactions with many clients. We also minimize bandwidth by making our PDP scheme check for proof of data possession without retrieving entire file blocks. A deterministic guarantee of possession can be given by when the client asks for proof for all the file blocks. But we give probabilistic guarantee of possession wherein we let our scheme access a random subset of the file blocks from the server's storage to compute the proof. The computation complexity at the client is not of much importance. Table 1 shows the computation details of the proposed scheme in comparison with S-PDP given by [4]. All exponentiations and multiplications listed in the table are modular operations.

When the input file F has n=1,000,000 blocks, assuming that the server S deletes at least 1% of F, the client C can detect server misbehavior with probability over 99% by asking proof for c = 460 randomly selected blocks.

When the client asks the server for a proof for a challenge with c=460 randomly selected blocks out of a file n=1,000,000 blocks, the S-PDP scheme requires the server to perform 461 exponentiations, 919 multiplications and 459 additions while the ∑-PDP needs only 1380 multiplications and 1377 additions. We eliminate any exponentiation in our scheme. To check the proof at the client, S-PDP needs to perform 462 exponentiations, 461 hashes, 460 inverses and 460 additions while ∑-PDP uses only 5 exponentiations, 923 multiplications and 918 additions.

Table 1. Computations involved in S-PDP and ∑-PDP

	S-PDP	**∑-PDP**
client computation to pre-process a file	$2n$ exponentiations, n multiplications, n hashes	$2n$multiplications, $2n$ additions
server computation (to generate proof)	$(c+1)$ exponentiations, $(2c-1)$ multiplications, $(c-1)$ additions, 1 hash	$(3c)$ multiplications, $(3(c-1))$ additions
client computation (to verify the proof)	$(c+2)$ exponentiations, $(c+1)$ hash, c inverses, c multiplications	5 exponentiations, $(2c+3)$ multiplications, $2(c-1)$ additions
Server block access	$O(1)$	$O(1)$
communication	$O(1)$	$O(1)$
Client storage	$O(1)$	$O(1)$

We are interested in minimizing the server computation and hence we have very few multiplications and additions in order to generate proof of possession while the S-PDP scheme proposed by Ateniese $et\ al.$ [4] needs exponentiations, hash operations along with the multiplication and addition operations to be done in order to generate the proof. Exponentiations can be expensive and we eliminate any such computation at the server. This shows that the proposed scheme is more efficient.

We perform multiplications on mod q while S-PDP has modulus N which results in larger number of multiplications. For comparison sake we first convert modular exponentiations to modular multiplications using square and multiply presented in [23]. The number of modular multiplications is equal to the number of $1's$ in the binary representation of the exponent.

Therefore, the total number of modular multiplications is at least the number of bits and at most twice the number of bits. If we consider the signed binary representation of the exponent, then the number of $1's$ will be $\frac{1}{3}$ times the total number of bits. For example, if the exponent is 1024-bits, then the number of 1's in that number would be $\frac{1}{3}(1024) = 342$. Therefore, the total number of modular multiplications involved in computing a modular exponentiation of a 1024-bit exponent is $1024 + 342 = 1366$. Table 4.2 gives the total number of modular multiplications involved in online computation of the client and the server in the proposed scheme in comparison with the $S - PDP$ for one message block.

N=1024 bits, p=160 bits, q=160 bits, m=1024 bits, d = 1024 bits, a_i=1024 bits, s=1024 bits, e=1024 bits, M=1024 bits.

The sizes of the parameters considered in the above computations are as follows:

For comparison between the two schemes, mod N multiplications need to be represented in terms of mod p/ mod q. The time taken for a mod n operation is proportional to l^2, where *l* is the number of bits in *n*. Taking *q* to have 160 bits and *N* to have 1024 bits, the time taken for a mod q operation is proportional to (160 * 160) and the time taken for a mod N operation is proportional to (1024 * 1024). Therefore, the number of mod q operations in a mod N operation is $(\frac{1024*1024}{160*160})$ = 40.96.

The small number of fixed exponentiations involved in ∑-PDP shows the improvement over the number of exponentiations that are linear in the number of challenges in S-PDP.

Table 2. Modular multiplications involved in ∑-PDP in comparison with S-PDP

<table>
<tr><td></td><td colspan="2">S-PDP</td><td colspan="2">∑-PDP</td></tr>
<tr><td>To generate tag/signature (Offline computations at client)</td><td colspan="2">$(h(W_1).g^m))^d \ mod \ N$

g^{md} : $\left(\frac{1}{3} * 2048\right) + 2048$ = 2730 multiplications
$(h(W_1).g^m))^d$: 1.5 * 2730 * n = 4095n multiplications
Total number of mod N multiplications: 4095n</td><td colspan="2">$\sigma_{1,m_1} = (z_{11}, z_{12})$
$= (m_1 w_1 + v_{11} \ mod \ q, m_1 w_2 + v_{12} \ mod \ q)$
Total number of mod q multiplications: 2n</td></tr>
<tr><td>To generate proof (Online computations at the server)</td><td colspan="2">$T = T_{i_1,m_{i_1}}^{a_1} \ mod \ N$
$T_{i_1,m_{i_1}}^{a_1} * \ldots * T_{i_c,m_{i_c}}^{a_c}$: 1366 * c multiplications
$g_s^{a_1,m_{i_1}+\cdots+a_c,m_{i_c}}$: 4096 multiplications
Total number of mod N multiplications: 4096+1366c</td><td colspan="2">$M = a_1 m_{i_1} mod \ q$
$(s_1, s_2) = a_1 z_{11} mod \ q, a_1 z_{12} mod \ q$

Total number of mod q multiplications: 3c</td></tr>
<tr><td>To check proof (Online and offline computations at the client)</td><td>Online computations</td><td>Offline computations</td><td>Online computations</td><td>Offline computations</td></tr>
<tr><td></td><td>$\tau = T^e$:1366 multiplications
$\frac{\tau}{(h(W_1))^{a_1}}$: c multiplications
$H(\tau^s \ mod \ N)$: 1366 multiplications

Total number of mod N multiplications : 2732+c</td><td>c number of $(h(W_1))^{a_1}$:1366 c
(mod N)</td><td>$g_1^{s_1} g_2^{s_2} = V h^M$
$g_1^{s_1}$:($\frac{1}{3}$*160)+160 = 213 multiplications
$g_1^{s_1} g_2^{s_2}$: 1.5*213 = 320 multiplications
h^M: 1366 multiplications
Vh^M: 1 multiplication

Total number of mod p multiplications: 1687</td><td>$V = g_1^{(b_1 v_{11} + b_2 v_{12} + \ldots. + b_c v_{1c})} * g_2^{(b_1 v_{21} + b_2 v_{22} + \ldots. + b_c v_{2c})}$:
Total number of mod p multiplications: 320+2c</td></tr>
</table>

4.6.1. Computations to generate tags/signatures at the client (offline computations)

The following table gives data obtained by varying the size of input message blocks, *n* in generating tags/signatures at the client. The same is represented graphically in Figure 4. Note that X-axis is n and Y-axis is S-PDP and ∑-PDP.

Table 3. Multiplications to generate tags/signatures at client

	S-PDP (4095n)	**Σ –PDP (2n)**	
n	mod N multiplications	mod q multiplications	mod p multiplications (1 mod p multiplication = 40.96 mod q multiplications)
100	409500	200	4.88
250	1023750	500	12.2
500	2047500	1000	24.4
750	3071250	1500	36.6
1000	4095000	2000	48.8
2500	10237500	5000	122
5000	20475000	10000	244
7500	30712500	15000	366
10000	40950000	20000	488
25000	102375000	50000	1220
50000	204750000	100000	2440
75000	307125000	150000	3660
100000	409500000	200000	4880
250000	1023750000	500000	12200
500000	2047500000	1000000	24400
750000	3071250000	1500000	36600
1000000	4095000000	2000000	48800
2500000	10237500000	5000000	122000
5000000	20475000000	10000000	244000

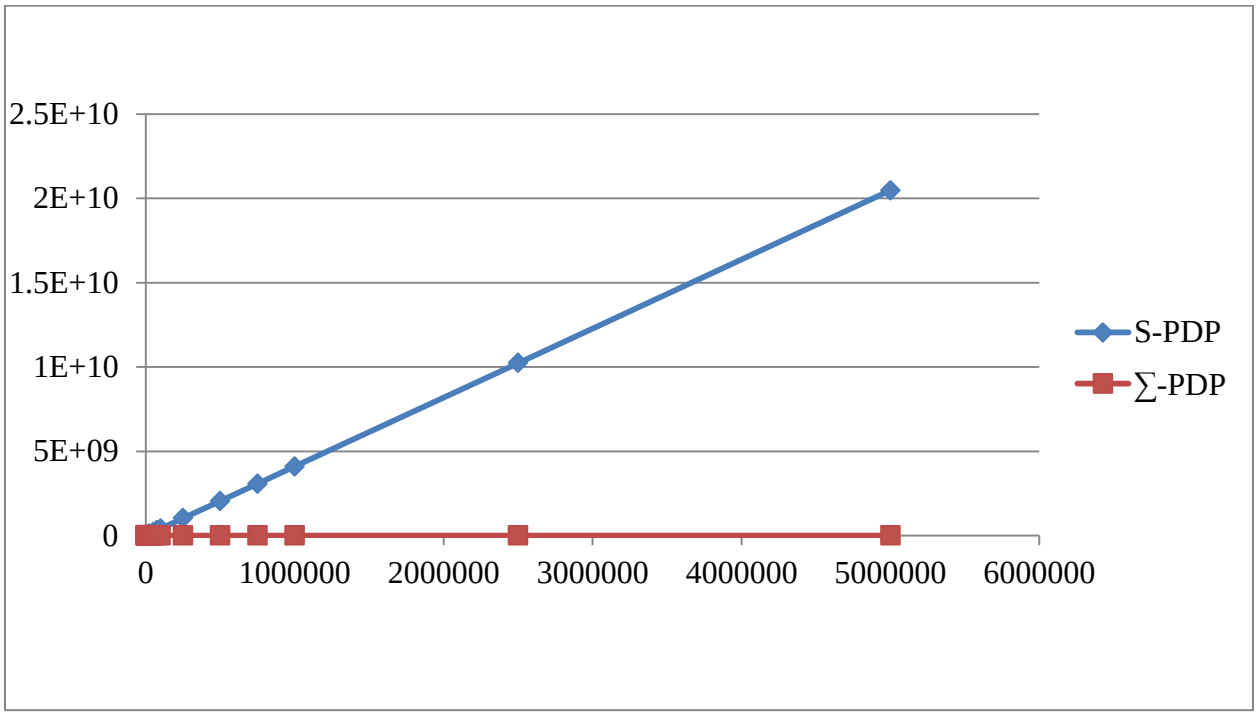

Figure 4. Multiplications at client to generate tags/signatures

4.6.2. Computations to generate proof at server (online computations)

The following table gives data obtained by varying the size of challenge blocks, *c* in generating proof at the server. The same is represented graphically in Figure 5 where X-axis is c and Y-axis is S-PDP and ∑-PDP.

Table 4. Multiplications to generate proof at server

	S-PDP (4096+1366c)	Σ –PDP (3c)	
c	mod N multiplications	mod q multiplications	mod p multiplications (1 mod p multiplication = 40.96 mod q multiplications)
10	17756	30	0.732422
15	24586	45	1.098633
20	31416	60	1.464844
25	38246	75	1.831055
30	45076	90	2.197266
35	51906	105	2.563477
40	58736	120	2.929688
45	65566	135	3.295898
50	72396	150	3.662109
75	106546	225	5.493164
100	140696	300	7.324219
150	208996	450	10.98633
200	277296	600	14.64844
250	345596	750	18.31055
300	413896	900	21.97266
350	482196	1050	25.63477
400	550496	1200	29.29688
450	618796	1350	32.95898
500	687096	1500	36.62109

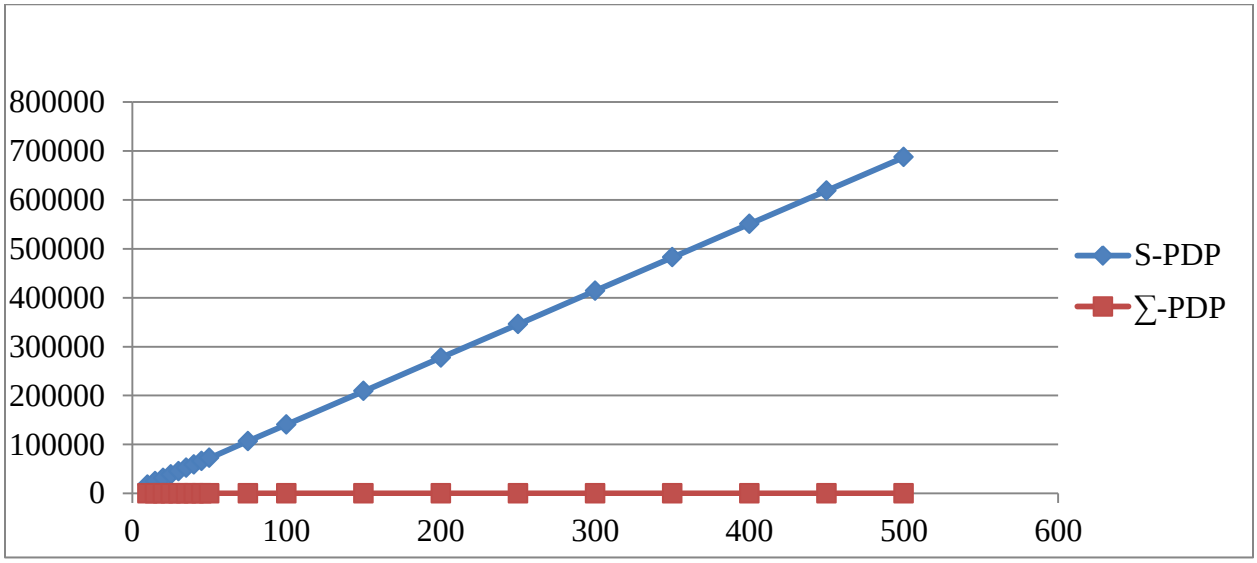

Figure 5. Multiplications at the server to generate proof

4.6.3. Computations to check proof at the client on c challenge blocks

This section gives details on computations involved in checking proof at the client.

4.6.3.1. Online multiplications at server to generate proof

The following table gives data obtained by varying the size of challenge blocks, *c* in checking proof at the client. The same is represented graphically in Figure 6 where X-axis is c and Y-axis is S-PDP and ∑-PDP.

Table 5. Online multiplications to check proof at the client

c	**S-PDP (1366c)**	**Σ –PDP (320+c)**
	mod N multiplications	mod p multiplications
10	13660	330
15	20490	335
20	27320	340
25	34150	345
30	40980	350
35	47810	355
40	54640	360
45	61470	365
50	68300	370
75	102450	395
100	136600	420
150	204900	470
200	273200	520
250	341500	570
300	409800	620
350	478100	670
400	546400	720
450	614700	770
500	683000	820

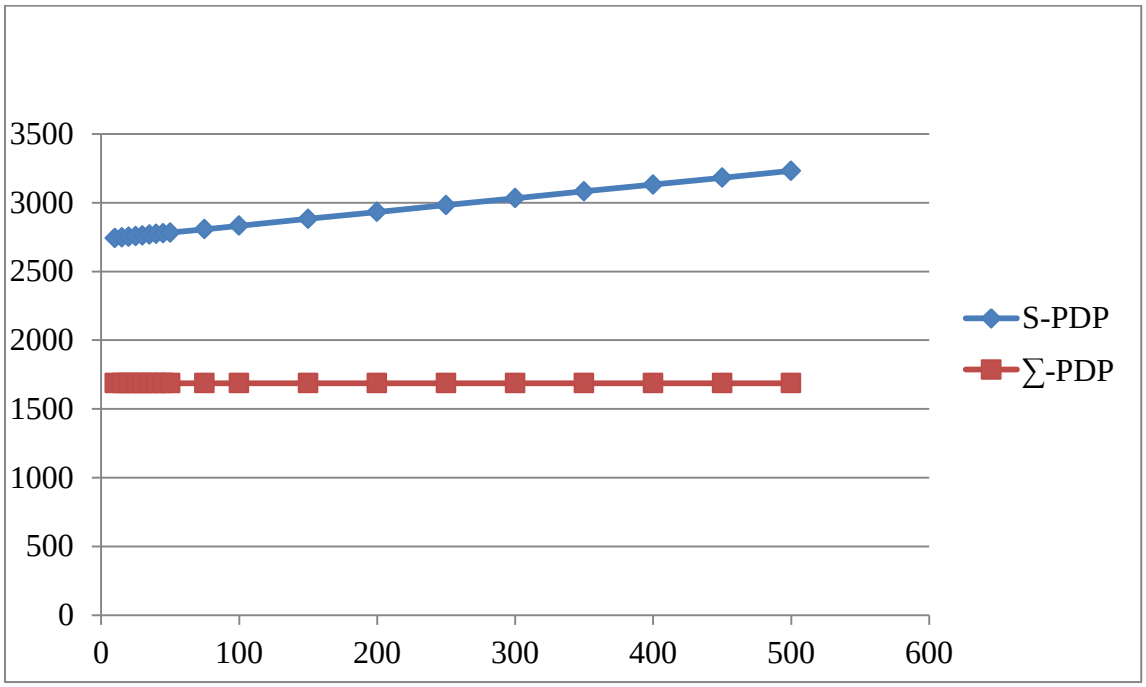

Figure 6. Online multiplications at client to check proof

4.6.3.2. Offline multiplications at server to generate proof

The following table gives data obtained by varying the size of challenge blocks, *c* in checking proof at the client. The same is represented graphically in Figure 7. Note that X-axis is c and Y-axis is S-PDP and ∑-PDP.

Table 6. Offline multiplications to check proof at the client

c	S-PDP (1366c)	Σ –PDP (320+c)
	mod N multiplications	mod p multiplications
10	13660	330
15	20490	335
20	27320	340
25	34150	345
30	40980	350
35	47810	355
40	54640	360
45	61470	365
50	68300	370
75	102450	395
100	136600	420
150	204900	470
200	273200	520
250	341500	570
300	409800	620
350	478100	670
400	546400	720

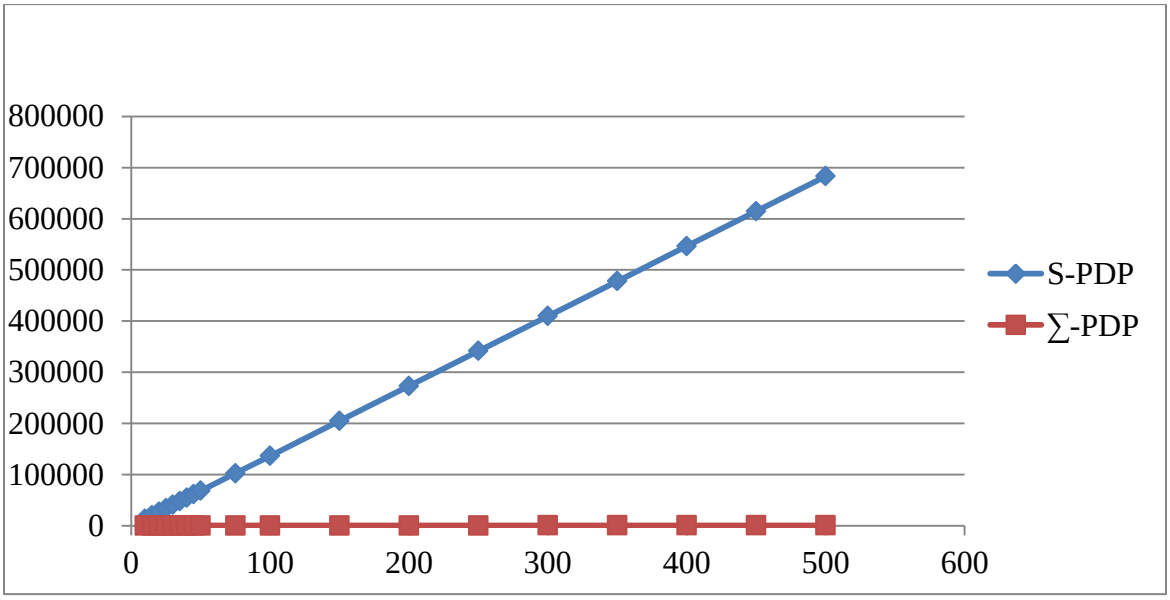

Figure 7. Offline computations to check proof at client

4.6.4. Comparison of ∑-PDP with previous work

In this section, Proof of Storage (POS) [35] is compared with PDP. The major difference between POS and PDP is that, PDP concentrates on reducing the computation complexity on the server due to limited bandwidth and also due to the huge number of requests that the server receives from multiple clients while the goal of POS is to reduce the communication complexity, mostly to make server-to-client communication independent of file length. This is done to reduce network traffic and overhead on both the client and server. The client-to-server communication is large in both PDP and POS.

In POS, the server computation is a constant multiple of the file size. In order to homomorphically combine the tags on all the input messages the verifier will need to access and compute a single tag on all the messages. This makes it linear in file length. Since this happens at the server, this has been made independent of the file length in ∑-PDP.

4.6.5. Computations involved in ∑-PDP as compared to POS

The major difference between Proofs of Storage (POS) from Homomorphic Identification Protocols and ∑- PDP scheme is that, we concentrate on reducing the computation complexity on the server due to limited bandwidth and also due to the huge number of requests that the server receives from multiple clients while their goal is to reduce the communication complexity both client-to-server and server-to-client and most importantly to make server-to-client independent of file length. This is done to reduce network traffic and overhead on both the client and server. In Proof of Storage (POS), a pseudorandom function, F, that takes as input a security key k that is given by the client is used at the server in order to generate the challenge. This is done to reduce client-to-server communication, so instead of the client generating the challenge and then sending it to the server to generate proof on it, the server itself generates the challenge

using a pseudorandom function, given key k. This poses security questions as the key k is made public.

Another observation made for comparison sake in POS is that, although the challenge vector $\vec{c}$ sent by the client is a random integer, the challenge length is taken to be equal to the input file length n. The challenge length in ∑- PDP is a random subset of input length n which makes the proof probabilistic and efficient.

POS has modulo N operations where N = p.q and p and q are primes. On the other hand, ∑- PDP consists of mod p and mod q operations. In POS, client generates n number of tags on n input message blocks. The following computations are done in the scheme:

Offline computations at the client:

$Combine_1(\vec{c}, \vec{\alpha}) = \prod_{i=1}^{n} \alpha_i^{c_i} mod\ N$: This takes n number of exponentiations and n multiplications.

Online computations at the server to generate proof:

$Combine_3(\vec{c}, \vec{\gamma}) = \prod_{i=1}^{n} \gamma_i^{c_i} mod\ N$: Takes n exponentiations and (n-1) number of multiplications, where $i = n$

$\mu_i = \sum_i c_i f_i$: Takes n multiplication.

Therefore, the server takes a total of 2n exponentiations and n multiplications in order to generate proof of storage.

CHAPTER 5. IMPLEMENTATION

Each of the cryptographic algorithms designed in the proposed scheme are implemented using Crypto++, the C++ class library of cryptographic algorithms and schemes by Wei Dai. The experiment was performed on an Intel 2.30 GHz i7 core. The machine runs Windows 7 Enterprise edition. The algorithms use Crypto++, a crypto class library version 5.6.1. Crypto++ helps perform difficult mathematical operations and cryptographic algorithms involved in the proposed algorithms. Microsoft Visual Studio 2012 was used to compile the C++ source code.

The five polynomial-time algorithms $(KeyGen, Sign, Ver, GenProof, CheckProof)$ are implemented and the details of the implementation are given in this Section.

$KeyGen$: This algorithm generates the public and private keys that are used throughout the scheme. It takes as input a security parameter k, generators g_1 and g_2, witness (w_1, w_2) and returns a pair of public and secret key (pk_i, sk).

As per the details of the proposed scheme, the algorithm uses pseudorandom functions in order to generate the witness (w_1, w_2) and the security parameters $(v_{11}, v_{21}), (v_{12}, v_{22}), \dots, (v_{1n}, v_{2n})$ for $1 \leq i \leq n$, which constitute the secret key $sk = ((w_1, w_2), ((v_{11}, v_{21}), (v_{12}, v_{22}), \dots, (v_{1n}, v_{2n})))$. Since the pseudorandom number generator (the function PRNG) does not allow us to control the input (key) to the function so as to generate the same outputs at a later point, we compromise this at the implementation stage and hence use AES (Advanced Encryption Standard) in CBC mode in order to generate the secret parameters.

Dhparam generates safe primes *p*, *q* such that $p = nq + 1$ of size 1024 and 1023 respectively. This function also generates generator g of sub group of order 1023. To generate a cipher text of size 64 characters(256 bits), an encryption key of length 16 hex characters(64 bits)

and 16 hex characters(64 bits) initial vector length is used on a plain text of length 24 characters(96 bits). The first 32 characters (128 bits) of the cipher text is used as the witness w_1 with 0x(32), the next 32 are used as w_2. The same is repeated with different cipher text to get the secret parameters $(v_{11}, v_{21}), (v_{12}, v_{22}), \dots, (v_{1n}, v_{2n})$ for $1 \leq i \leq n$.

The function ModularExponentiation() is used to perform exponentiations under modular p throughout the scheme. Similarly, modular additions are performed using the corresponding function. For 10 message blocks, a simple rand() function is used to generate the indices i_j and co-efficients b_j. Modular multiplications and modular additions are performed to check the equality of the proof of possession.

5.1. Timing Analysis for Varying Message and Challenge Lengths

Experiments were performed for varying input message and challenge length. Table 6. gives results for various combinations of message and challenge length. The data is analyzed for max input file length = 35 messages and challenge size = 20. The sizes of parameters used in the implementation is given as: Size of witnesses (w's): 128 bits, size of secret parameters (v's): 128 bits, size of messages (m's): 152 bits.

The offline timing analysis includes preparing the message blocks prior to computation and also generating one time signatures on each of the message blocks. The time taken for online computation remains same if the number of challenge blocks is increased because of the fixed number of exponentiations involved in checking the proof.

Table 7. Time to perform computations for varying file and challenge length

Length of F=n blocks	Length of challenge=c blocks	Offline computations	Online computations
15	10	1 ms	0.72 ms
20	10	1.09 ms	0.78 ms
30	10	1.29 ms	0.81 ms
35	10	1.34 ms	0.87 ms
35	15	1.34 ms	0.87 ms

CHAPTER 6. CONCLUSION AND FUTURE RESEARCH

This chapter talks about the conclusion of this thesis study and scope for suture research in this topic.

6.1. Main Contribution and Results

We introduced a scheme for ∑-Provable Data Possession, in which we minimize client and server computation, file block access complexity, and client-server communication complexity by a major reduction in the number of exponentiations involved in the proof of possession. Our solution minimizes the storage at both client and server. We completely eliminate exponentiation which can be expensive at the server to generate proof of possession. We instead perform simple modular multiplications and additions at the server. This provides unlimited number of verification of the proof of possession. We allow the client to verify data possession with a minimum of 5 exponents irrespective of the size of the challenge while the previous solution requires a minimum of c exponentiations. The complexity analysis under various levels are presented to compare the results of this thesis with the previous PDP solutions. Proof of security was proved based on discrete logarithm assumption over finite group. Finally, the algorithms designed in the scheme are efficiently implemented using Crypto++, a class library for C++ using Microsoft Visual Studio 2012 to check for the proof of possession. The scheme was successfully implemented to get the desired results.

We reduced the number of exponentiations involved in the proof. This minimized client and server computation- we relieve the server of computational overhead client-server communication complexity storage at both client and server.

6.2. Limitations and Future Research

One of the main areas that we did not address in this thesis work is public verifiability. In the "$CheckProof$" phase, for a challenge e, given the public key $pk \leftarrow (h, V)$ along with the signature on that block, any third party can verify the validity of the proof of possession. This will require the third party to interact with the client in order to get the public key pk for that challenge set. Future researchers can include the study of the client giving this information to any third party that requests it, making the scheme publicly verifiable.

Future research can also include having multiple servers in the PDP scenario while also keeping in mind the complexity factor which was the main focus of this thesis.

BIBLIOGRAPHY

[1] S. Kamara, "Computing Securely with Untrusted Resources," 2008.

[2] N. Leavitt, "Is cloud computing really ready for prime time," in *IEEE Computer Society*, 2009.

`[3] C. Erway, A. Küpçü, C. Papamanthou and R. Tamassi, "Dynamic provable data possession," in *Proceedings of the 16th ACM Conference on Computer and Communication Security*, 2009.

[4] G. Ateniese , R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson and D. Song, "Provable Data Possession at Untrusted Stores," in *Proceedings of the 14th ACM, CCS*, 2007.

[5] A. Fiat and A. Shamir, "How to prove yourself. Practical solutions to identification and signature problems," Advances in Cryptology-Crypto, Springer-Verlag, 1986.

[6] S. Goldwasser, S. Micali and C. Rackoff, "The knowledge complexity of interactive proof-systems," in *ACM Symposium on Theory of Computing (STOC)*, 1985.

[7] A. Juels and B. S. Kaliski, "PORs: Proofs of retrievability for large files," in *Proceedings of 14th ACM Conference on Computer and Communications Security(CCS '07)*, 2007.

[8] W. Diffie, Interviewee, "*How Secure Is Cloud Computing."* [Interview], 2009.

[9] C. Schnorr, "Efficient signature generation by smart cards," *Journal of Cryptology,* 1991.

[10] G. Ateniese, R. D. Pietro, L. V. Mancini and G. Tsudik, "Scalable and efficient provable data possession," in *Proceedings on the 14th International conference on Security and Privacy in Communication networks (SecureComm '08)*, 2008.

[11] R. Curtmola, O. Khan and R. Burns, "Robust remote data checking," in *The 4th International workshop on storage security and survivability (StorageSS '08)*, 2008.

[12] Y. Zhu, H. Wang, Z. Hu, G. Ahn, H. Hu and S. Yau, "Efficient provable data possession for hybrid clouds," in *The 17th ACM Conference on computer and communications security (CCS '10)*, 2010.

[13] G. Ateniese, R. Burns, R. Curtmola and J. Herrin, "Remote data checking using provable data possession," in *ACM Transactions on Information and System Security (TISSEC)*, 2011.

[14] Y. Lindell, "Lecture notes on Introduction to Cryptography," 2006. [Online]. Available: http://u.cs.biu.ac.il/~lindell/89-656/Intro-to-crypto-89-656.pdf.

[15] I. Damgard, "Lecture Notes on ∑-protocols, Aarhus University," Department of Computer Science (DAIMI), 2011.

[16] C. Hazay and Y. Lindell, "Efficient Secure Two-Party Protocol," Springer, 2010.

[17] R. Cramer, I. Damgard and J. Nielsen, "Lecture notes on Electronic Payment Systems," Cryptographic Protocol Theory, 2009.

[18] Y. Lindell and J. Katz, "Introduction to modern cryptography," Chapman & Hall/CRC Press, 2007.

[19] O. Goldreich, S. Goldwasser and S. Micali, "How to construct random functions," in *the IEEE Symposium on the Foundations of Computer Science (FOCS '84),IEEE Computer Society*, 1984.

[20] J. Hastad, R. Impagliazzo, L. Levin and M. Luby, "A pseudorandom generator from any one-way function," in *SIAM Journal on Computing*, 1999.

[21] A. Mohan and R. Katti, "Provable Data Possession using Sigma-protocols," in *Proceedings of 11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications(TRUSTCOM '12), IEEE Computer Society press*, 2012.

[22] M. Bellare and O. Goldreich, "On defining proofs of knowledge," in *The Proceedings of 12th Annual International Cryptology Conference in Advances of Cryptology (CRYPTO '92)*, 1992.

[23] D. R. Stinson, "Cryptography Theory and Practice," University of Waterloo: Chapman & Hall/CRC, 2006.

[24] G. Ateniese , R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson and D. Song, "Provable Data Possession at Untrusted Stores," in *in Proceedings of the 14th ACM, CCS*, 2007.

[25] S. Goldwasser, S. Micali and C. Rackoff, "The knowledge complexity of interactive proof-

systems.," in *ACM Symposium on Theory of Computing (STOC)*, 1985.

[26] A. Juels and B. Kaliski, "PORs: Proofs of retrievability for large files," in *ACM CCS*, 2007.

[27] W. Diffie, "Interviewee, *How Secure Is Cloud Computing?."* [Interview]. 2009.

[28] G. Ateniese, R. D. Pietro, L. V. Mancini and G. Tsudik, "Scalable and efficient provable data possession," in *SecureComm*, 2008.

[29] R. Curtmola, O. Khan and R. Burns, "Robust remote data checking," in *4th International workshop on storage security and survivability, StorageSS*, 2008.

[30] Y. Zhu, H. Wang, Z. Hu, G. Ahn, H. Hu and S. Yau, "Efficient provable data possession for hybrid clouds," in *17th ACM Conference on computer and communications security, ACM, CCS*, 2010.

[31] M. Bellare and O. Goldreich, "On defining proofs of knowledge," in *Proceedings of CRYPTO*, 2009.

[32] N. Leavitt, "Is cloud computing really ready for prime time," in *IEEE Computer Society*, 2009.

[33] Y. Lindell, "Introduction to Cryptography," 2006.

[34] A. Mohan and R. Katti, "Provable Data Possession using Sigma-protocols," in *Proceedings of 11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications(TRUSTCOM), IEEE Computer Society press*, 2012.

[35] G. Ateniese, S. Kamara and J. Katz, "Proof of Storgae from Homomorphic Identification Protocols," in *Advances in Cryptology-ASIACRYPT '09*, 2009.

APPENDIX A. SOURCE CODE

A.1. Code to Generate Witnesses, Secret Parameters and Input Message Blocks

```
int main(int argc, char* argv[])
{
AutoSeededRandomPool prng;
byte key[AES::DEFAULT_KEYLENGTH]= {0xA7, 0x8D, 0xC8, 0xBE, 0x6C, 0xE3,
0xA0, 0x9A, 0x7A, 0x0B, 0x2A, 0x92, 0x51, 0x03, 0x19, 0x6D};
byte iv[AES::BLOCKSIZE]= {0x0D, 0xF6, 0xD8, 0x71, 0x90, 0xA3, 0x42, 0xD5, 0x25,
0x18, 0xED, 0x4E, 0x7B, 0xBE, 0xD3, 0x7B};
string plain = "1234567891234567";
string cipher, encoded, recovered;
encoded.clear();
StringSource(key, sizeof(key), true,
new HexEncoder(
        new StringSink(encoded)
        ) ; // encoded is the encryption key
    encoded.clear();
    StringSource(iv, sizeof(iv), true,
        new HexEncoder(
            new StringSink(encoded)
        )
    );
    try
```

```
        {
                string str= plain; // plain is the plain text to get witness w
                CBC_Mode< AES >::Encryption e;
                e.SetKeyWithIV(key, sizeof(key), iv);
                StringSource s(plain, true,
                        new StreamTransformationFilter(e,
                                new StringSink(cipher)
                        )
                );
#if 0
                StreamTransformationFilter filter(e);
                filter.Put((const byte*)plain.data(), plain.size());
                filter.MessageEnd();
                const size_t ret = filter.MaxRetrievable();
                cipherh.resize(ret);
                filter.Get((byte*)cipherh.data(), cipherh.size());
#endif
        }
        catch(const CryptoPP::Exception& e)
        {
                cerr << e.what() << endl;
                exit(1);
        }
```

```
encoded.clear();
StringSource(cipher, true,
        new HexEncoder(
                new StringSink(encoded)
                string str= encoded;  // str is the cipher text to get witness w
                        );
```

A.2. Code to Generate Signatures with Modular Exponentiation Multiplication Function

```
string strprep1, strprep2;
strprep1 = str.substr (0,32; // sub-string 0-32 from cipher text for w1
strprep2 = str.substr (32,32); // sub-string 33- 64 from cipher text for w2
string strw1, strw2;
const char * cw1 = {strprepw1.c_str()};// convert string to char
const char * cw2 = {strprepw2.c_str()};
Integer ulw1(cw1); // convert char to Integer
Integer ulw2(cw2);
Integer h1 = ModularExponentiation(g1,ulw1, p); // g1^w1 mod p
Integer h2 = ModularExponentiation(g2,ulw2, p); //g2^w2 mod p
Integer h= a_times_b_mod_c(h1, h2, p); // h = g1^w1 g2^w2 mod p
```

A.3. Code to Generate Challenge and Proof on this Challenge

```
int k=1, n[10], l[20];
Integer M= 0;
Integer BZ1= 0, BZ2= 0;
Integer bz1[10], bz2[10];
```

```
for(k=1; k<10; k++)
{
    srand (k);
    n[k] = rand() % 11; // indices k
    l[k] = rand() % 21; co-efficients
    Integer ulchal1 = a_times_b_mod_c(*ulm1[n[k]], l[k], q);
    M = (M+ ulchal1) % q; // M
    bz1[k] = a_times_b_mod_c(sigma11[n[k]], l[k], q);
    bz2[k] = a_times_b_mod_c(sigma12[n[k]], l[k], q);
    BZ1 = (BZ1 + bz1[k]) % q; // s_1
    BZ2 = (BZ2 + bz2[k]) % q; // s_2
}
```

A.4. Code to Check Proof

```
Integer LHS_CHECKPROOF_1 = ModularExponentiation(g1, BZ1, p);
Integer LHS_CHECKPROOF_2 = ModularExponentiation(g2, BZ2, p);
Integer LHS_CHECKPROOF = a_times_b_mod_c(LHS_CHECKPROOF_1,
LHS_CHECKPROOF_2, p); // g_1^{s_1} g_2^{s_2}: LHS of checkproof
  Integer biv1i[20], biv2i[20];
Integer BIV1I= 0, BIV21;
for(k=1; k<10; k++)
{
    srand (k);
    n[k] = rand() % 11;
```

```
        l[k] = rand() % 21;
        biv1i[i] = a_times_b_mod_c(*ulv1[n[k]], l[k], q);
        BIV11 = (BIV11 + biv1i[i]) % q; //b_1 v_11 + b_2 v_12 + .... + b_c v_1c mod q
        biv2i[i] = a_times_b_mod_c(*ulv2[n[k]], l[k], q);
        BIV21 = (BIV21 + biv2i[i]) % q; // (b_1 v_21 + b_2 v_22 + .... + b_c v_2c mod q
}
        Integer RHS_CHECKPROOF_1 =  ModularExponentiation(g1,BIV1I, p); //
g_1^(b_1 v_11 + b_2 v_12 + .... + b_c v_1c) mod p
        Integer RHS_CHECKPROOF_2 =  ModularExponentiation(g2,BIV2I, p); //
g_2^(b_1 v_21 + b_2 v_22 + .... + b_c v_2c) mod p
        Integer RHS_CHECKPROOF1 = a_times_b_mod_c(RHS_CHECKPROOF_1,
RHS_CHECKPROOF_2, p);
        Integer RHS_CHECKPROOF_3 = ModularExponentiation(h,M, p);// h^M mod p
        Integer RHS_CHECKPROOF = a_times_b_mod_c(RHS_CHECKPROOF1,
RHS_CHECKPROOF_3, p); // Vh^M mod p
        cout<< "The LHS of check proof :  " << LHS_CHECKPROOF << " ? "<< "The
RHS of check proof  :" << RHS_CHECKPROOF << endl;
getch();
        return 0;
}
```