BRICS DS-03-9 M. J. Jurik: Extensions to the Paillier Cryptosystem with Applications to Cryptological Protocols

Basic Research in Computer Science

Extensions to the Paillier Cryptosystem with Applications to Cryptological Protocols

Mads J. Jurik

BRICS Dissertation Series **DS-03-9**

ISSN 1396-7002 **August 2003**

Copyright © 2003, Mads J. Jurik.
BRICS, Department of Computer Science
University of Aarhus. All rights reserved.

Reproduction of all or part of this work is permitted for educational or research use on condition that this copyright notice is included in any copy.

See back inner page for a list of recent BRICS Dissertation Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

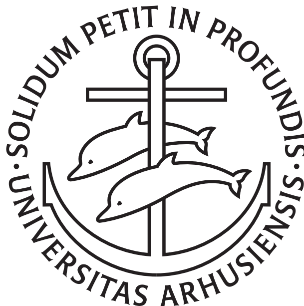
BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `DS/03/9/`

Extensions to the Paillier Cryptosystem with Applications to Cryptological Protocols

Mads Johan Jurik

PhD Dissertation



Department of Computer Science
University of Aarhus
Denmark

Extensions to the Paillier Cryptosystem with Applications to Cryptological Protocols

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfilment of the Requirements for the
PhD Degree

by
Mads Johan Jurik
March 19, 2004

Abstract

The main contribution of this thesis is a simplification, a generalization and some modifications of the homomorphic cryptosystem proposed by Paillier in 1999, and several cryptological protocols that follow from these changes.

The Paillier cryptosystem is an additive homomorphic cryptosystem, meaning that one can combine ciphertexts into a new ciphertext that is the encryption of the sum of the messages of the original ciphertexts. The cryptosystem uses arithmetic over the group $\mathbb{Z}_{n^2}^*$ and the cryptosystem can encrypt messages from the group $\mathbb{Z}_n$. In this thesis the cryptosystem is generalized to work over the group $\mathbb{Z}_{n^{s+1}}^*$ for any integer $s > 0$ with plaintexts from the group $\mathbb{Z}_{n^s}$. This has the advantage that the ciphertext is only a factor of $(s+1)/s$ longer than the plaintext, which is an improvement to the factor of 2 in the Paillier cryptosystem. The generalized cryptosystem is also simplified in some ways, which results in a threshold decryption that is conceptually simpler than other proposals. Another cryptosystem is also proposed that is length-flexible, i.e. given a fixed public key, the sender can choose the s when the message is encrypted and use the message space of $\mathbb{Z}_{n^s}$. This new system is modified using some El Gamal elements to create a cryptosystem that is both length-flexible and has an efficient threshold decryption. This new system has the added feature, that with a globally setup RSA modulus n, provers can efficiently prove various relations on plaintexts inside ciphertexts made using different public keys.

Using these cryptosystems several multi-party protocols are proposed:

- A mix-net, which is a tool for making an unknown random permutation of a list of ciphertext. This makes it a useful tool for achieving anonymity.
- Several voting systems:
  - An efficient large scale election system capable of handling large elections with many candidates.
  - Client/server trade-offs: 1) a system where vote size is within a constant of the minimal size, and 2) a system where a voter is protected even when voting from a hostile environment (i.e. a Trojan infested computer). Both of these improvements are achieved at the cost of some extra computations at the server side.
  - A small scale election with perfect ballot secrecy (i.e. any group of persons only learns what follows directly from their votes and the final result) usable e.g. for board room election.

- A key escrow system, which allows an observer to decrypt any message sent using any public key set up in the defined way. This is achieved even though the servers only store a constant amount of key material.

The last contribution of this thesis is a petition system based on the modified Weil pairing. This system greatly improves the naive implementations using normal signatures from using an order of $\mathcal{O}(tk)$ group operations to using only $\mathcal{O}(t+k)$, where t is the number of signatures checked, and k is the security parameter.

Acknowledgements

First I would like to thank my supervisor Ivan B. Damgård for creating a research environment that has been very inspiring. He always had time to discuss new ideas and give feedback on early ideas and research problems.

I would also like to thank the cryptology group at BRICS for listening to all my seminars and providing valuable feedback on the presentations, and especially to Louis Salvail for making the whole experience a lot funnier than expected.

A big thanks also goes to Dan Boneh for hosting my stay abroad at Stanford University and to the Ph.D. students at the Security Lab in the Stanford Computer Science Department, for making the stay both a professional and social success.

Jan Camenish and Berry Schoenmakers also deserves thanks for providing alot of insightfull comments on the thesis during the defence. This has helped improve the overall quality of the thesis.

Finally I would like to thank Kirill Morozov, Anne Grethe Jurik and Bolette Ammitzbøll Madsen for proof reading this dissertation, thereby helping to reduce the number of errors.

Mads Johan Jurik,
Århus, March 19, 2004.

Contents

Abstract **v**

Acknowledgements **vii**

1 Introduction **1**
- 1.1 Motivation 1
- 1.2 List of Publications 2
  - 1.2.1 A Generalization of Paillier's Public-Key System with Applications to Electronic Voting 2
  - 1.2.2 Client/Server Tradeoffs for Online Elections 2
  - 1.2.3 A Length-Flexible Threshold Cryptosystem with Applications 3
  - 1.2.4 A Key-Escrow Public-key Cryptosystem 3
- 1.3 Overview of Chapters 3

2 Improving the Paillier Cryptosystem **7**
- 2.1 Introduction 7
  - 2.1.1 Background 7
  - 2.1.2 Related Work 8
  - 2.1.3 Contribution 8
- 2.2 A Generalization of Paillier's Probabilistic Encryption Scheme . 9
  - 2.2.1 Security 13
  - 2.2.2 Adjusting the Block length 16
- 2.3 Some Optimizations and Implementation Issues 17
  - 2.3.1 An Alternative Encryption Function 17
  - 2.3.2 Optimizations of Encryption and Decryption 18
- 2.4 Some Building Blocks 21
  - 2.4.1 A Threshold Variant of the Scheme 21
  - 2.4.2 Some Auxiliary Protocols 25
- 2.5 Introducing an El Gamal Element 29
  - 2.5.1 Security of the Cryptosystem 30
- 2.6 An Efficient Length-Flexible Threshold Cryptosystem 33
- 2.7 A Proof Friendly Variant 34
  - 2.7.1 Security of the Threshold Cryptosystems 34
  - 2.7.2 Proofs in the Proof Friendly Variant 35
  - 2.7.3 Homomorphic Properties 41

3 Anonymity Using Mix-nets **43**
3.1 Introduction 43
3.1.1 Background 43
3.1.2 Contribution 44
3.2 The Mix-net Model 44
3.3 Adversaries 45
3.4 Security of the Mix-net 46
3.5 The System 46
3.6 Security Proofs 50

4 Secure On-line Voting **55**
4.1 Introduction 55
4.1.1 Background 55
4.1.2 Related Work 58
4.1.3 Contribution 59
4.2 Efficient Electronic Voting 61
4.2.1 Model and Notation 61
4.2.2 A Yes/No Election 62
4.2.3 A Multi-Candidate Election 63
4.2.4 A variant with smaller vote size 63
4.3 Client/Server Trade-Offs 66
4.3.1 The Minimal Vote Election Scheme 66
4.3.2 An Alternative System 70
4.3.3 Protecting Clients Against Hackers 73
4.3.4 Interval Proofs for Paillier Encryptions 76
4.4 Self-Tallying Elections with Perfect Ballot Secrecy 79
4.4.1 Setup Phase 81
4.4.2 Ballot Casting 83
4.4.3 Tallying 86
4.4.4 Efficiency Comparison to Scheme from [47] 88

5 Key Escrow **89**
5.1 Introduction 89
5.1.1 Background 89
5.1.2 Contribution 90
5.2 Model 91
5.3 A Simple Key Escrow System 92
5.4 Threshold Key Escrow 93
5.4.1 Removing the Trusted Third Party 95
5.5 Encryption Verification 95
5.6 Improving Performance for $s > 1$ 96
5.7 Security of the System 97

6 Efficient Petitions **99**
6.1 Introduction 99
6.1.1 Background 99
6.1.2 Related Work 99

6.1.3 Contribution 100
6.2 Model 100
6.2.1 Properties for Petitions 101
6.3 Standard Petitions 101
6.4 Threshold Petitions 102
6.5 An Efficient Petition System 103
6.5.1 Aggregate Signatures 103
6.5.2 Using Aggregating Signatures 104
6.5.3 Security of Aggregate Signatures 105
6.5.4 An Aggregate Signature System 107
6.5.5 Petitions based on Aggregate Signatures 109
6.6 Comparison 110

Bibliography **113**

Chapter 1

Introduction

I may not have gone where I intended to go, but I think I have ended up where I intended to be.

— Douglas Adams

1.1 Motivation

One of the main topics of this thesis is length-flexibility. This is the ability to extend the plaintext space at encryption time rather than at key generation time, when the public key is chosen. In the literature this has been available for symmetric ciphers (where sender and receiver both know the key). In public key systems it is available by splitting the plaintext into blocks and encrypting these blocks separately. A typical way to construct a length-flexible system with the public key properties is to make an encryption of a symmetric key using the public key system, and then use that symmetric key to make a symmetric encryption of the message itself.

Another main topic is homomorphic encryption in which it is possible to combine two encryptions to get an encryption that encrypts some combination of the two original plaintexts. The most useful variant of these is the additive homomorphisms, where the new encryption is the sum of the two original plaintexts. Systems with additive homomorphisms are useful for a lot of cryptological protocols like electronic voting, multi-party computation, etc., providing that they also support verification protocols.

There is also the question of threshold decryption. In multi-party computation models, all the actual computations are made inside encryptions using the homomorphic properties of the specific cryptosystem. However, this implies that to get the result someone has to decrypt at some point. This is where threshold decryption is used. It allows servers to decrypt if less than half the servers are trying to cheat or disrupt the protocol.

All these topics are interesting by themselves, but the real interesting things start to happen when they are combined. This leads to an additive homomorphic length-flexible threshold cryptosystem. This enables one to set up a system once and for all, and start doing computation on problems of various sizes. This is especially true in case of electronic voting where the same system might be

used to hold a local election and a national election. These two forms of election have very different sizes and in normal cryptosystems it would be necessary to create two different cryptosystems, or one would risk that the local election could become very inefficient because it is using a cryptosystem designed for a national election. This is where length-flexibility comes in handy, in the sense that now the plaintext space can be adjusted to create both the elections in a setting that is essentially as efficient as possible for both problems.

1.2 List of Publications

1.2.1 A Generalization of Paillier's Public-Key System with Applications to Electronic Voting

The paper *A Generalization of Paillier's Public-Key System with Applications to Electronic Voting*, was published as a technical report [25] and in the proceedings of a conference [26]. It is accepted in a journal and is currently waiting to be printed [31].

[25] I. Damgård, and M. Jurik: *A Generalisation, a Simplification and some Applications of Paillier's Probabilistic Public-Key System*, BRICS Report Series, record 2000/45. BRICS, `http://www.brics.dk/Publications/`, December 2000.

[26] I. Damgård, and M. Jurik: *A Generalisation, a Simplification and some Applications of Paillier's Probabilistic Public-Key System*, Public Key Cryptography (PKC 2001), LNCS 1992, pp. 119-136. Springer Verlag, 2001.

[31] I. Damgård, M. Jurik, and J.B. Nielsen: *A Generalization of Paillier's Public-Key System with Applications to Electronic Voting*, to appear in special issue on Financial Cryptography, International Journal on Information Security (IJIS). Springer Verlag.

The original technical report was rewritten for the proceedings version, which have afterwards replaced the original technical report. The journal version has been combined with an unpublished result of Damgård and Nielsen [33]. The main differences are several implementation details, and the voting section is extended with a technique to handle a general number of candidates in one of the voting schemes and not only powers of 2 as in the conference version.

1.2.2 Client/Server Tradeoffs for Online Elections

The paper *Client/Server Tradeoffs for Online Elections* was published at a conference [27].

[27] I. Damgård, and M. Jurik: *Client/Server Tradeoffs for Online Elections*, Public Key Cryptography (PKC 2002), LNCS 2274, pp. 125-140. Springer Verlag, 2002.

1.2.3 A Length-Flexible Threshold Cryptosystem with Applications

The paper *A Length-Flexible Threshold Cryptosystem with Applications* was published as a technical report [28] and accepted at a conference [29].

[28] I. Damgård, and M. Jurik: *A Length-Flexible Threshold Cryptosystem with Applications*, BRICS Report Series, record 2003/16. BRICS, `http://www.brics.dk/Publications/`, March 2003.

[29] I. Damgård, and M. Jurik: *A Length-Flexible Threshold Cryptosystem with Applications*, to appear in Information Security and Privacy (ACISP 2003), LNCS. Springer Verlag, 2003.

The conference version is a shorter version of the technical report. The technical report has a couple of alternative cryptosystems, some optimization observations and proofs of the various theorems.

1.2.4 A Key-Escrow Public-key Cryptosystem

This paper has only been published as a technical report [30] and have not been submitted for any conference yet:

[30] I. Damgård, and M. Jurik: *Scalable Key-Escrow*, BRICS Report Series, record 2003/22. BRICS, `http://www.brics.dk/Publications/`, May 2003.

1.3 Overview of Chapters

The papers mostly concern new variants of the Paillier cryptosystem and their application, so this dissertation has been split up in a chapter with the variants of the Paillier system and some chapters with various cryptological protocols based on these variants. This is done to avoid reinventing the wheel in every chapter and has led to the following chapters:

Chapter 2: This chapter covers the Paillier cryptosystem and the simplification, generalization and modification of the cryptosystem. This leads to several new cryptosystems with various nice properties. The sections on

- Simplifying the encryption and decryption
- Making a threshold version
- Using n^{s+1} as modulus instead of n^2
- Length-flexible encryption

are based on the paper *A Generalization of Paillier's Public-Key System with Applications to Electronic Voting* [31].

The second part of the chapter is about length-flexible threshold cryptosystems. This improves on the length-flexible version proposed in the

first part of the chapter at the cost of introducing a new assumption. The problem with the first threshold version is that it sometimes requires a multi-party computation to update the secret keys. The new length-flexible threshold system is achieved by introducing an El Gamal like element in the encryption. The resulting threshold system has some protocols to prove the correctness of the encryptions. It even allows a person sending and receiving messages to prove relations on the messages inside these encryptions. This even holds for messages encrypted using different public keys. The system requires that an RSA-modulus with unknown factorization is set up in advance. Assuming a trusted third party for generating this modulus is more feasible than usual, since this can be achieved by using a trusted hardware box to create the modulus. After the modulus has been created the hardware box can be destroyed to ensure that no information on the factorization can be gained from it. This is based on the paper *A Length-Flexible Threshold Cryptosystem with Applications* [28].

Chapter 3: In this chapter a mix-net is proposed. It has several nice properties (these will be further explained in the chapter):

- ***Length-flexibility***
- ***Length-invariance***
- ***Provable security***
- ***Universal verifiability***
- ***Strong correctness***
- ***Order flexibility***

This is the first scheme that has all the above attributes, although this comes at the cost of efficiency compared to hybrid mixes [4, 46]. This is essentially done by exchanging the El Gamal cryptosystem in [1] with one of the length-flexible systems from chapter 2. This is one of the applications presented in the paper *A Length-Flexible Threshold Cryptosystem with Applications* [28].

Chapter 4: This chapter contains 3 different flavors of voting.

The first type of voting system is based on the homomorphic properties in one of the Paillier cryptosystems presented in chapter 2. This leads to an election protocol which greatly improves previous voting protocols, especially in the case where a large number of candidates and voters are participating. It uses zero-knowledge protocols to create a proof of correctness of the ballot that has size $\mathcal{O}(\log(L)\max(k, L\log(M)))$, where L is the number of candidates, M the number of voters, and k is the security parameter. All valid votes can be combined using the homomorphic property of the cryptosystem and the complete result can be decrypted using the threshold decryption. This is based on the voting system in the paper *A Generalization of Paillier's Public-Key System with Applications to Electronic Voting* [31].

The next voting system addresses 2 problems:

Minimizing voter load: The voting system mentioned above has the problem that the workload of the voters can become quite large, if there is a lot of candidates and voters. In essence, what we could hope to achieve is that the voter only has to provide something of size $\mathcal{O}(\log(L))$.

Hostile environments: Sometimes voters have to vote in hostile environments. This could be a computer at some library or a home computer infested with Trojans/worms. This means that some adversary can monitor or even take over the voting ability of the voter.

The first point is addressed by making 2 system, one that is within a constant of absolute minimal size of a vote $\mathcal{O}(\max(\log(L), k))$ and one that has a larger vote size of $\mathcal{O}(\log(L)\max(L, k))$. To do this, the servers have to transform the votes into what would have been valid votes in the first voting system in this chapter. Then the combination and decryption of the result can be done by following the same procedure. The second system does not achieve minimal votes, but the amount of work performed by the servers will be significantly less in most realistic scenarios.

To secure a voter against hostile environments we assume he receives a permutation in some secure way, which the adversary cannot learn (i.e. at registration). Then the voter can submit the permuted value of his choice of candidate, which is transformed at the server side in a similar manner to the minimal vote system. This results in a system, where the adversary controlling the hostile environment will gain no information on the vote. If the adversary takes over the computer, the best it can do is to submit a vote for a random candidate. This is still a serious attack, but an improvement over total control of which candidate the vote is for. This is based on the paper *Client/Server Tradeoffs for Online Elections* [27].

The final voting system presented is a self-tallying, dispute-free voting system with perfect ballot secrecy. This type of voting system was introduced by Kiayias and Yung in [47]. We will present a way to use one of the cryptosystems in chapter 2 to make a more efficient system, that is also feasible for multi-candidate elections. This is based on one of the applications presented in the paper *A Length-Flexible Threshold Cryptosystem with Applications* [28].

Chapter 5: Two classes of cryptosystems were presented in chapter 2. First there was the generalization of the Paillier cryptosystem. This was used to create the second class by adding an El Gamal element. The encryption function in these two classes are related by this equation:

$$E'(m, r) = (g^r \bmod n, E(m, h^r \bmod n))$$

This means, that there are two kinds of secret keys, namely the secret key for the E' encryption and the secret key for the E encryption. Both the

secret key for E and E' can be used to decrypt the ciphertext and recover m.

In this chapter, we present a key escrow system that uses both of these keys such that a surveillance agency can decrypt messages without revealing the secret key of the receiving party. The idea is that the key escrow servers hold the key for the E encryption, and the users of the system generate different keys for the E' encryption. This will allow the escrow servers to decrypt any messages encrypted under the E' encryption function with only one secret key - namely the secret key for E encryption function. This is based on the paper *Scalable Key-Escrow* [30].

Chapter 6: This chapter concerns the creation of an electronic petition system. Two of the three system presented in this chapter are based on standard techniques. 1) a system using standard signatures, 2) a system using threshold signatures, and 3) a system using the new notion of aggregate signatures.

The first two systems follow directly from standard techniques and are included to provide a frame of reference. The third system uses the notion of aggregate signatures which will be presented in the chapter. Aggregate signatures have the property that two signatures can be combined into a new signature. This new signature can be checked with a public key, that can be computed from the two public keys under which the original signatures could be verified.

We provide an example of a system that supports aggregate signatures based on the modified Weil pairing. This system has the nice properties that it can combine two signatures or two public keys using just one addition over the elliptic curve. Checking a signature can be done using $\mathcal{O}(k)$ additions on the elliptic curve, where k is the security parameter.

This leads to a petition system where a verifier just has to combine t keys and check one signature. This requires $\mathcal{O}(t + k)$ additions on an elliptic curve, which is very efficient compared to the $\mathcal{O}(tk)$ additions used when checking t standard signatures on an elliptic curve.

This is based on unpublished results made in cooperation with Dan Boneh, Ben Lynn and Hovav Shacham. These results were never published due to a concurrent, but independent result by Boldyreva [13], that covered the cryptological building blocks in the result.

Chapter 2

Improving the Paillier Cryptosystem

Every day you may make progress. Every step may be fruitful. Yet there will stretch out before you an ever-lengthening, ever-ascending, ever-improving path. You know you will never get to the end of the journey. But this, so far from discouraging, only adds to the joy and glory of the climb.
— Winston S. Churchill.

Paillier proposed a new cryptosystem in [52]. It is an efficient and additively homomorphic cryptosystem, which makes it useful for a lot of things including for instance voting (although some additional building blocks are needed). In this chapter we will look at ways to simplify, generalize and change the original cryptosystem to make it even more useful. Some of the properties achieved in this chapter are an efficient threshold decryption, length-flexibility (i.e. given a fixed key any size of plaintext can be encrypted and decrypted), and how to make zero knowledge proofs over different public keys. During this chapter we will see several cryptosystems that each have their advantages, and in the following chapters some of these will be used to improve different cryptological protocols.

2.1 Introduction

2.1.1 Background

In [52], Paillier proposes a new probabilistic encryption scheme based on computations over the group $\mathbb{Z}_{n^2}^*$, where n is an RSA modulus. This scheme has some very attractive properties, in that it is homomorphic, allows encryption of many bits in one operation with a constant expansion factor, and allows efficient decryption. This makes it potentially interesting for many cryptological protocols such as electronic voting and mix-nets.

This scheme is related to an earlier cryptosystem by Okamoto and Uchiyama [51], in which the group $\mathbb{Z}_{p^2q}^*$ is used, where p and q are large primes. The main difference is that the homomorphic property of this scheme requires that the sum of the messages being added is smaller than p, which is unknown. The Paillier scheme does not suffer from this problem since the homomorphic computations are simply calculated modulo n.

El Gamal proposed a cryptosystem that makes computations in the group $\mathbb{Z}_p^*$ using a generator g of a subgroup of size q (where $q|p-1$). The good thing about this is that the order q of g is publicly known, which allows a simple threshold decryption by simply making a Shamir secret sharing [54] of the secret over the group $\mathbb{Z}_q$. Another advantage of this cryptosystem is that given the public parameters p, q, g anyone can set up their own secure cryptosystem.

2.1.2 Related Work

In work independent from, but earlier than the result from this chapter, Fouque, Poupard and Stern [37] proposed the first threshold version of Paillier's original scheme. Like the first threshold scheme in this chapter, [37] uses an adaptation of Shoup's threshold RSA scheme [57], but apart from this the techniques are somewhat different, in particular because we construct a threshold version for our generalized cryptosystem (and not only Paillier's original scheme).

2.1.3 Contribution

In this chapter we propose a generalization of Paillier's scheme using computations modulo n^{s+1}, for any integer $s \geq 1$. We also show that the system can be simplified (without degrading security) such that the public key can consist of only the modulus n. This allows instantiating the system such that the block length for the encryption can be chosen freely for each encryption, independently of the size of the public key (length-flexible) and without losing the homomorphic property. The generalization also allows reducing the expansion factor from 2 for Paillier's original system to almost 1. We prove that the generalization is as secure as Paillier's original scheme. We also provide a number of ways to optimize both the encryption and decryption operations, in particular a new algorithm for encryption which, compared to a naive implementation of Paillier's original scheme, saves a factor of 4 in computing time. In general, it saves a factor of $4s$ compared to a straightforward implementation of the generalized system.

We propose a threshold variant of the generalized system, allowing a number of servers to share knowledge of the secret key, such that any large enough subset of them can decrypt a ciphertext, while smaller subsets have no useful information. We prove in the random oracle model that the scheme is as secure as a standard centralized implementation.

We also propose a zero-knowledge proof of knowledge allowing a prover to show that a given ciphertext encodes a given plaintext. From this we derive other tools, such as a protocol showing that a ciphertext encodes one out of a number of given plaintexts. Finally, we propose a protocol that allows verification of multiplicative relations among encrypted values without revealing extra information.

The threshold decryption of this system, however, has a couple of problems: 1) the decryption servers have to perform work and store key material proportional to the size of messages they decrypt and 2) threshold decryption in the length-flexible case sometimes need a heavy multi-party computation (when a

message longer than the currently stored key arrives). To fix this we make a combination of the generalized Paillier and the El Gamal cryptosystem. This results in a new cryptosystem that inherits the homomorphic property from the Paillier system and some of the flexibility from the El Gamal cryptosystem. It is secure under the Paillier (DCRA) and the composite DDH (DDH over an RSA moduli) assumptions, or - at a moderate loss of efficiency - based only on the Paillier assumption. It is also related to a Paillier-based scheme presented in [23], but is more efficient and is also length-flexible.

We achieve two new properties. First, our scheme allows several users to use the same modulus. This makes it possible to do efficient zero-knowledge proofs for relations between ciphertexts created under different public keys.

Secondly, we propose a threshold decryption protocol where keys can be set up so that messages of arbitrary length can be handled efficiently with the same (fixed size) keys. In addition, the computational work done by each server does not depend on the message length, only the cost of a final public post-processing is message dependent. This is in contrast to the system mentioned above where the threshold decryption requires work from the servers proportional to the size of the message encrypted and in some cases a heavy multi-party computation. We also give efficient zero-knowledge protocols for proving various claims on encrypted values in this cryptosystem.

2.2 A Generalization of Paillier's Probabilistic Encryption Scheme

The public-key cryptosystem we describe here uses computations modulo n^{s+1}, where n is an RSA modulus and s is a natural number. It contains Paillier's scheme [52] as a special case by setting $s = 1$.

Consider a modulus $n = pq$, p, q odd primes, where $\gcd(n, \varphi(n)) = 1$. When p, q are large and randomly chosen, this will be satisfied except with negligible probability. Such an n will be called *admissible* in the following. For such an n, $\mathbb{Z}^*_{n^{s+1}}$ as a multiplicative group is a direct product $G \times H$, where G is cyclic of order n^s and H is isomorphic to $\mathbb{Z}^*_n$. This follows directly from the Chinese Remainder Theorem and the fact that $\mathbb{Z}^*_{p^{s+1}}$ is cyclic of order $(p-1)p^s$. Thus, the factor group $\bar{G} = \mathbb{Z}^*_{n^{s+1}}/H$ is also cyclic of order n^s. For an arbitrary element $a \in \mathbb{Z}^*_{n^{s+1}}$, we let $\bar{a} = aH$ denote the element represented by a in the factor group $\bar{G}$.

Lemma 2.1 *For any admissible n and $s < p, q$, the element $n + 1$ has order n^s in $\mathbb{Z}^*_{n^{s+1}}$.*

Proof. Consider the integer $(1+n)^i = \sum_{j=0}^{i} \binom{i}{j} n^j$. This number is 1 modulo n^{s+1} for some i if and only if $\sum_{j=1}^{i} \binom{i}{j} n^{j-1}$ is 0 modulo n^s. Clearly, this is the case if $i = n^s$, so it follows that the order of $1 + n$ is a divisor in n^s, i.e., it is a number of form $p^\alpha q^\beta$, where $\alpha, \beta \leq s$. Set $a = p^\alpha q^\beta$, and consider a term $\binom{a}{j} n^{j-1}$ in the sum $\sum_{j=1}^{a} \binom{a}{j} n^{j-1}$. We claim that each such term is divisible by a: this is trivial if $j > s$, and for $j \leq s$, it follows because $j!$ cannot have p

or q as prime factors, and so a must divide $\binom{a}{j}$. Now assume for contradiction that $a = p^\alpha q^\beta < n^s$. Without loss of generality, we can assume that this means $\alpha < s$. We know that n^s divides $\sum_{j=1}^{a} \binom{a}{j} n^{j-1}$. Dividing both numbers by a, we see that p must divide the number $\sum_{j=1}^{a} \binom{a}{j} n^{j-1}/a$. However, the first term in this sum after division by a is 1, and all the rest are divisible by p, so the number is in fact 1 modulo p, and we have a contradiction. □

Since the order of H is relatively prime to n^s, the above lemma implies immediately that the element $\overline{1+n} := (1+n)H \in \bar{G}$ is a generator of $\bar{G}$, except possibly for $s \geq p, q$. So the cosets of H in $\mathbb{Z}^*_{n^{s+1}}$ are

$$H, (1+n)H, (1+n)^2 H, ..., (1+n)^{n^s-1} H$$

which leads to a natural numbering of these cosets. The following lemma captures the structure of $\mathbb{Z}^*_{n^{s+1}}$ in a more concrete way:

Lemma 2.2 *For any admissible n and $s < p, q$, the map $\psi_s : \mathbb{Z}_{n^s} \times \mathbb{Z}^*_n \to \mathbb{Z}^*_{n^{s+1}}$ given by $(x, r) \mapsto (1+n)^x r^{n^s} \bmod n^{s+1}$ is an isomorphism, where $\psi_s(x_1 + x_2 \bmod n^s, r_1 r_2 \bmod n) = \psi_s(x_1, r_1)\psi_s(x_2, r_2) \bmod n^{s+1}$.*

Proof. Let $\pi : \mathbb{Z}^*_n \to \mathbb{Z}^*_{n^{s+1}}$ be given by $r \mapsto \psi_s(0, r) = r^{n^s} \bmod n^{s+1}$. By the above enumeration of the cosets of H it is enough to prove that $\pi(r_1 r_2 \bmod n) = \pi(r_1)\pi(r_2) \bmod n^{s+1}$ and that π maps $\mathbb{Z}^*_n$ injectively to H. First, it is clear that $\pi(r) \in H$. By looking at the binomial expansion it is easy to see that $r^{n^s} \equiv (r+n)^{n^s} \bmod n^{s+1}$. This proves the homomorphic property directly and by the pigeon hole principle implies that π is injective. □ *Proof.* Let $\pi : \mathbb{Z}^*_n \to \mathbb{Z}^*_{n^{s+1}}$ be given by $r \mapsto \psi_s(0, r) = r^{n^s} \bmod n^{s+1}$. By the above enumeration of the cosets of H it is enough to prove that $\pi(r_1 r_2 \bmod n) = \pi(r_1)\pi(r_2) \bmod n^{s+1}$ and that π maps $\mathbb{Z}^*_n$ injectively to H. First, it is clear that $\pi(r) \in H$. By looking at the binomial expansion it can be verified that $r^{n^s} \equiv (r+n)^{n^s} \bmod n^{s+1}$:

$$(r+n)^{n^s} = r^{n^s} + \binom{n^s}{1} r^{n^s-1} n + \cdots + \binom{n^s}{s} r^{n^s-s} n^s = r^{n^s} \bmod n^{s+1}$$

All the terms except r^{n^s} gets a factor of n^s from the binomial and a factor of atleast one n from the expansion of $(r+n)$ and thus they cancel out modulo n^{s+1}. This proves the homomorphic property of π. □

This lemma gives us the following encoding of the cosets: $(\overline{1+n})^i = \psi_s(i, \mathbb{Z}^*_n)$.

The final technical observation we need is that ψ can be inverted given the factorization of n. In particular, taking discrete logarithms base $\overline{n+1}$ in $\bar{G}$ is easy given the factorization.

Theorem 2.1 *For any admissible n and $s < p, q$, the map $\psi_s : \mathbb{Z}_{n^s} \times \mathbb{Z}^*_n \to \mathbb{Z}^*_{n^{s+1}}$ given by $(x, r) \mapsto (1+n)^x r^{n^s} \bmod n^{s+1}$ can be inverted in polynomial time given $\lambda(n)$, the least common multiple of $p-1$ and $q-1$.*

Proof. We first show how to find i from $(1+n)^i \bmod n^{s+1}$ without using λ. If we define the function L by $L(b) = (b-1)/n$ then clearly we have

$$L((1+n)^i \bmod n^{s+1}) = (i + \binom{i}{2} n + ... + \binom{i}{s} n^{s-1}) \bmod n^s$$

We now describe an algorithm for computing i from this number.

The general idea of the algorithm is to extract the value part by part, so that we first extract $i_1 = i \bmod n$, then $i_2 = i \bmod n^2$ and so forth (in the following we'll use $i_j = i \bmod n^j$ to denote these values). It is easy to extract $i_1 = L((1+n)^i \bmod n^2) = i \bmod n$. Now we can extract the rest by the following induction step: In the j'th step we know i_{j-1}. This means that $i_j = i_{j-1} + k * n^{j-1}$ for some $0 \leq k < n$. If we use this in

$$L((1+n)^i \bmod n^{j+1}) = (i_j + \binom{i_j}{2} n + ... + \binom{i_j}{j} n^{j-1}) \bmod n^j$$

we can notice that each term $\binom{i_j}{t+1} n^t$ for $j > t > 0$ satisfies that $\binom{i_j}{t+1} n^t = \binom{i_{j-1}}{t+1} n^t \bmod n^j$. This is because the contributions from $k * n^{j-1}$ vanish modulo n^j after multiplication by n. It follows that:

$$L((1+n)^i \bmod n^{j+1}) = (i_j + \binom{i_{j-1}}{2} n + ... + \binom{i_{j-1}}{j} n^{j-1}) \bmod n^j$$

Then we can isolate i_j

$$i_j = L((1+n)^i \bmod n^{j+1}) - (\binom{i_{j-1}}{2} n + ... + \binom{i_{j-1}}{j} n^{j-1}) \bmod n^j$$

This equation leads to the following algorithm where a has the form $(n+1)^i$:

```
dLog_s(a):
begin
  i := 0;
  for j := 1 to s do        /* i = i_{j-1} */
  begin
    t_1 := L(a mod n^{j+1});
    t_2 := i;
    for k := 2 to j do      /* t_2 = i(i-1)···(i-k+2) */
    begin
      i := i - 1;
      t_2 := t_2 * i mod n^j;
      t_1 := t_1 - (t_2 * n^{k-1})/k! mod n^j;     /* t_1 = t_1 - (i choose k) n^{k-1} */
    end
    i := t_1;
  end
  return i;
end
```

Assume now that we are given $c = (1+n)^i r^{n^s} \bmod n^{s+1}$. We show how to find i and r given λ. To find i compute

$$c^\lambda = (1+n)^{i\lambda \bmod n^s} r^{n^s \lambda \bmod n^s \lambda} = (1+n)^{i\lambda \bmod n^s} \ .$$

Then using the above algorithm find $i\lambda \bmod n^s$ and extract i. Now compute $r^{n^s} = c(1+n)^{-i} \bmod n^{s+1}$ and compute a such that $a\lambda + 1 = 0 \bmod n^s$. This is possible because $\gcd(\lambda, n^s) = 1$. Then

$$(r^{n^s})^{\frac{a\lambda+1}{n^s}} \bmod n = r^{a\lambda+1} \bmod n = (r^\lambda)^a r \bmod n = r \bmod n = r \ .$$

□

We are now ready to describe our cryptosystem. In fact, for each natural number s, we can build a cryptosystem CS_s, as follows:

Key Generation: Given the security parameter k as input, choose an admissible RSA modulus $n = pq$ of length k bits[1]. Also choose an element $g \in \mathbb{Z}^*_{n^{s+1}}$ such that $g = (1+n)^j x \bmod n^{s+1}$ for a known j relatively prime to n and $x \in H$. This can be done, e.g. by choosing j, x at random first and computing g; some alternatives are described later. Let λ be the least common multiple of $p-1$ and $q-1$. By the Chinese Remainder Theorem, choose d such that $d \bmod n \in \mathbb{Z}^*_n$ and $d = 0 \bmod \lambda$. Any such choice of d will work in the following. In Paillier's original scheme $d = \lambda$ was used, which is the smallest possible value. However, when making a threshold variant, other choices are better - we expand on this in the following section.

Now the public key is $pk = (n, g)$ while the secret key is d.

Encryption: The plaintext set is $\mathbb{Z}_{n^s}$. Given a plaintext m, choose a random $r \in \mathbb{Z}^*_n$, and let the ciphertext be

$$E_{s,pk}(m, r) = g^m r^{n^s} \bmod n^{s+1}$$

Decryption: Given a ciphertext c, first compute $c^d \bmod n^{s+1}$. Clearly, if $c = E_{s,pk}(m, r)$, we get

$$c^d = (g^m r^{n^s})^d = ((1+n)^{jm} x^m r^{n^s})^d = (1+n)^{jmd \bmod n^s} (x^m r^{n^s})^{d \bmod \lambda}$$
$$= (1+n)^{jmd \bmod n^s}$$

Now apply the algorithm from the proof of theorem 2.1 to compute $jmd \bmod n^s$. Applying the same method on g clearly produces the value $jd \bmod n^s$, so this can either be computed on the fly or be saved as part of the secret key. In any case we obtain the plaintext by

$$(jmd) \cdot (jd)^{-1} = m \bmod n^s$$

[1]Strictly speaking, we also need that $s < p, q$, but this is insignificant since in practice, s will always be much smaller than p, q.

Clearly, this system is additively homomorphic over $\mathbb{Z}_{n^s}$, that is the product of encryptions of messages m, m' with the random values r, r' is an encryption of $m + m' \bmod n^s$ with the random value $r \cdot r' \bmod n$.

To facilitate comparison with Paillier's original system, we have kept the above system description as close as possible to that of Paillier. In particular, the description allows choosing g in a variety of ways. However, as we shall see, semantic security of the system is equivalent to a particular computational assumption, no matter how we choose g. In particular we may as well simplify matters and choose $g = n + 1$ always. This also allows for a more efficient implementation. Therefore, in the following sections, when we refer to CS_s, we usually mean the above system with $g = n + 1$.

2.2.1 Security

There are two basic flavors or strengths of security that one may consider, namely

- Is the scheme one-way, i.e. is it hard to compute the plaintext from the ciphertext?
- Is the scheme semantically secure, i.e. does any information at all about the plaintext leak, given the ciphertext?

We first give a short informal discussion on one-wayness, and then look at semantic security in more detail.

The homomorphic property of the scheme means that the problem of computing the plaintext from the ciphertext (and the public key) is random self-reducible: given any ciphertext c and public key n, g, one may choose $m \in \mathbb{Z}_{n^s}$, $j \in \mathbb{Z}^*_{n^s}$, $r, r' \in \mathbb{Z}^*_n$ at random and try to decrypt the ciphertext $c' = cg^m r^{n^s} \bmod n^{s+1}$ with respect to public key n, g' where $g' = g^j r'^{n^s} \bmod n^{s+1}$. If this succeeds, one can find the original plaintext by multiplying by j and subtracting m modulo n^s. Note that c', g' is a random ciphertext-generator pair, no matter how c, g were chosen. So any algorithm that can break a non-trivial fraction of the ciphertexts and choices of g can also break a random instance with significant probability. This motivates calling our scheme one-way if it is hard to find the plaintext given a random public key n, g and a random ciphertext c.

Proposition 2.1 *If for some t the scheme CS_t is one-way, then CS_s is one-way for any $s > t$. Especially CS_s is one-way for any s if Paillier's original scheme CS_1 is one-way.*

Proof. Assume that $s > t$ and that CS_t is one-way. Assume for the sake of contradiction that CS_s is not one-way. Then given a public key n, g and a ciphertext c_t from CS_t, we can transform this to a decryption problem in CS_s instead. Concretely, this means we consider c_t as a number modulo n^{s+1} (instead of n^{t+1}), and choose as the public generator a random number $\tilde{g} \in \mathbb{Z}^*_{n^{s+1}}$ such that $\tilde{g} \bmod n^{t+1} = g$. We then randomize c_t (modulo n^{s+1}) as described above. This produces a random instance of the decryption problem

in CS_s, so by assumption we can find the plaintext m in CS_s corresponding to c_t. We have of course that $m \in \mathbb{Z}_{n^s}$, and now clearly $m \bmod n^t$ is the plaintext corresponding to c_t in CS_t, so that CS_t is not one-way either. □

If we want to claim that a cryptosystem "hides" the plaintext in any reasonable sense, the one-way assumption is essentially the weakest possible assumption one can make. In [16], Catalano, Gennaro and Howgrave-Graham show that this assumption for CS_1 implies that one can make a *semantically secure* system hiding a logarithmic number of bits per ciphertext in the original system, and that a somewhat stronger assumption implies a system hiding a linear number of bits per ciphertext. It is easy to generalize these results to CS_s. However, none of the schemes constructed this way will be homomorphic.

The semantic security of our schemes can be based on the following assumption, introduced by Paillier in [52], the *decisional composite residuosity assumption* (DCRA):

Conjecture 2.1 *Let $\mathcal{A}$ be any probabilistic polynomial time algorithm, and assume $\mathcal{A}$ gets n, x as input. Here n has k bits and is chosen as described above. x is either random in $\mathbb{Z}^*_{n^2}$ or it is a random n'th power in $\mathbb{Z}^*_{n^2}$ (i.e. a random element in the subgroup H defined earlier). $\mathcal{A}$ outputs a bit b. Let $p_0(\mathcal{A}, k)$ be the probability that $b = 1$ if x is random in $\mathbb{Z}^*_{n^2}$, and $p_1(\mathcal{A}, k)$ the probability that $b = 1$, if x is a random n'th power. Then $\mid p_0(\mathcal{A}, k) - p_1(\mathcal{A}, k) \mid$ is negligible in k.*

Here, "negligible in k" means smaller than $1/f(k)$ for any polynomial f and all large enough k as usual.

We now discuss the semantic security of CS_s. There are several equivalent formulations of semantic security. We will use the following:

Definition 2.1 *An adversary $\mathcal{A}$ against a public-key cryptosystem gets the public key pk generated from security parameter k as input and outputs a message m. Then $\mathcal{A}$ is given an encryption under pk of either m or a message chosen uniformly in the message space, and outputs a bit. Let $p_0(\mathcal{A}, k)$ respectively $p_1(\mathcal{A}, k)$ be the probability that $\mathcal{A}$ outputs 1 when given an encryption of m respectively a random encryption. Define the* advantage *of $\mathcal{A}$ to be $Adv(\mathcal{A}, k) = |p_0(\mathcal{A}, k) - p_1(\mathcal{A}, k)|$. The cryptosystem is* semantically secure *if for any probabilistic polynomial time adversary $\mathcal{A}$, $Adv(\mathcal{A}, k)$ is negligible in k.*

In [52], Paillier showed that his cryptosystem (which is identical to our CS_1) is semantically secure if and only if DCRA holds. This holds for any choice of g, and follows easily from the fact that given a ciphertext c that is either random or encrypts a message m, we have $cg^{-m} \bmod n^2$ is either random in $\mathbb{Z}^*_{n^2}$ or a random n'th power. In particular, assuming DCRA, one may choose $g = n + 1$ always without degrading security. We now show that security of CS_s is equivalent to DCRA:

Theorem 2.2 *For any s, the cryptosystem CS_s is semantically secure if and only if the DCRA is true. This holds even if s is allowed to increase polynomially in the security parameter.*

Proof. From a ciphertext in CS_s, one can obtain a ciphertext in CS_1 by reducing modulo n^2, this implicitly reduces the message modulo n. It is therefore clear that if DCRA fails, then CS_s cannot be secure for any s.

For the converse, we assume that CS_s is not secure and we start by showing a relation between the security of CS_s and that of CS_t for values of $t < s$.

The message space of CS_s is $\mathbb{Z}_{n^s}$. Thus any message m can be written in radix n notation as an s-tuple $(m_s, m_{s-1}, ..., m_1)$, where each $m_i \in \mathbb{Z}_n$ and $m = \sum_{i=0}^{s-1} m_{i+1} n^i$. Let $\mathcal{D}_n(m_s, ..., m_1)$ be the distribution obtained by encrypting the message $(m_s, ..., m_1)$ under public key n. If one or more of the m_i are replaced by $*$'s, this means that the corresponding positions in the message are chosen uniformly in $\mathbb{Z}_n$ before encrypting.

Now, assume for simplicity that s is even, consider any adversary $\mathcal{A}$ against CS_s, and assume that $Adv(\mathcal{A}, k) \geq 1/f(k)$ for some polynomial f and infinitely many values of k. For any such value, we can assume without loss of generality, that we have $p_0(\mathcal{A}, k) - p_1(\mathcal{A}, k) \geq 1/f(k)$. Suppose we make a public key n from security parameter k, show it to $\mathcal{A}$, get a message $(m_s, ..., m_1)$ from $\mathcal{A}$ and show $\mathcal{A}$ a sample of $\mathcal{D}_n(*, ..., *, m_{s/2}, ..., m_1)$. Let $q(\mathcal{A}, k)$ be the probability that $\mathcal{A}$ now outputs 1. We must have

$$p_0(\mathcal{A}, k) - q(\mathcal{A}, k) \geq \frac{1}{2f(k)} \quad \text{or} \quad q(\mathcal{A}, k) - p_1(\mathcal{A}, k) \geq \frac{1}{2f(k)} \tag{2.1}$$

and one of these cases must be true for infinitely many values of k. In the first case in (2.1), we can make a successful adversary against $CS_{s/2}$, as follows: we get the public key n, show it to $\mathcal{A}$, get $(m_s, ..., m_1)$ as output from $\mathcal{A}$, and return $(m_s, ..., m_{1+s/2})$ as our output. We will get a ciphertext c that either encrypts $(m_s, ..., m_{1+s/2})$ in $CS_{s/2}$, or is a random ciphertext. If we consider c as an element in $\mathbb{Z}^*_{n^{s+1}}$, we know it is an encryption of some plaintext, which must have either $(m_s, ..., m_{1+s/2})$ or $s/2$ random elements in its least significant positions. Hence $c^{n^{s/2}} \bmod n^{s+1}$ is an encryption of $(m_s, ..., m_{1+s/2}, 0, ..., 0)$ or $(*, ..., *, 0, ..., 0)$. We then make a random encryption d of $(0, ..., 0, m_{s/2}, ..., m_1)$, give $c^{n^{s/2}} d \bmod n^{s+1}$ to $\mathcal{A}$ and return the bit $\mathcal{A}$ outputs. Now, if c encrypts $(m_s, ..., m_{1+s/2})$, we have shown to $\mathcal{A}$ a sample of $\mathcal{D}_n(m_s, ..., m_1)$, and otherwise a sample of $\mathcal{D}_n(*, ..., *, m_{s/2}, ..., m_1)$. So by assumption on $\mathcal{A}$, this breaks $CS_{s/2}$ with an advantage of $1/2f(k)$ for infinitely many k.

In the second case of (2.1), we can also make an adversary against $CS_{s/2}$, as follows: we get the public key n, show it to $\mathcal{A}$, and get a message $(m_s, ..., m_1)$. We output $(m_{s/2}, ..., m_1)$ and get back a ciphertext c that encrypts in $CS_{s/2}$ either $(m_{s/2}, ..., m_1)$ or something random. If we consider c as a number modulo n^{s+1}, we know that the corresponding plaintext in CS_s has either $(m_{s/2}, ..., m_1)$ or random elements in the least significant $s/2$ positions - and something unknown in the top positions. We make a random encryption d of $(*, .., *, 0, ..., 0)$, show $cd \bmod n^{s+1}$ to $\mathcal{A}$ and return the bit $\mathcal{A}$ outputs. If c encrypted $(m_{s/2}, ..., m_1)$, we have shown $\mathcal{A}$ a sample from $\mathcal{D}_n(*, .., *, m_{s/2}, ..., m_1)$, and otherwise a sample from $\mathcal{D}_n(*, ..., *)$. So again this breaks $CS_{s/2}$ with an advantage of $1/2f(k)$ for infinitely many k.

To sum up, we have: for any adversary $\mathcal{A}$ against CS_s, s even, there exists an adversary $\mathcal{A}'$ against $CS_{s/2}$, such that $Adv(\mathcal{A}', k) \geq 1/2f(k)$ for infinitely many

k. Similarly, for odd s, we can show existence of an adversary against either $CS_{(s+1)/2}$ or $CS_{(s-1)/2}$ with advantage at least $1/2f(k)$ for infinitely many k.

Repeated use of this result shows that for any adversary $\mathcal{A}$ against CS_s, there exists an adversary against CS_1 with advantage at least $1/2sf(k)$ for infinitely many k. Thus, since s is polynomially bounded as a function of k, CS_1 is not semantically secure, and this contradicts Paillier's original result. □

From the point of view of exact security analysis, one can note that from the proof above, it follows that the maximal advantage with which CS_s can be broken is at most a factor of $2s$ larger than the corresponding advantage for CS_1. Thus, there is no great security risk in using polynomially large values of s, if one believes that CS_1 is secure in the first place.

2.2.2 Adjusting the Block length

As mentioned, we may choose $g = n + 1$ always without losing security, and the public key may then consist only of the modulus n. This means that we can decide on a value for s at any point after the keys have been generated, or even let the sender decide on the fly when he encrypts a message. Concretely, the length-flexible system CS^* will work as follows:

Key Generation: Choose an admissible RSA modulus $n = pq$. Now the public key is $pk = n$ while the secret key is λ, the least common multiple of $(p - 1)$ and $(q - 1)$.

Encryption: Given a plaintext m represented as a non-negative integer, choose s such that $m < n^s$, choose a random $r \in \mathbb{Z}_n^*$, and let the ciphertext be

$$E^*_{pk}(m, r) = (1 + n)^m r^{n^s} \bmod n^{s+1}$$

Whenever a certain encryption is made it uses a fixed s and the derived encryption function will be denoted:

$$E^*_{(s),pk}(m, r) = (1 + n)^m r^{n^s} \bmod n^{s+1}$$

Decryption: Given a ciphertext $c = E^*_{pk}(m, r)$, compute $c^\lambda \bmod n^{s+1}$ (note that from the length of c, one can compute the correct value of s except with negligible probability). Clearly, if $c = E(m, r)$, we get

$$c^\lambda = ((1+n)^m r^{n^s})^\lambda = (1+n)^{m\lambda \bmod n^s} (r^{n^s})^{\lambda \bmod \lambda} = (1+n)^{m\lambda} \bmod n^{s+1}$$

Now apply the algorithm from theorem 2.1 to compute $m\lambda \bmod n^s$ and get the message by multiplying by λ^{-1} modulo n^s.

It is an immediate corollary to proposition 2.1 and theorem 2.2 that the above scheme is one-way, if CS_1 is one-way, respectively is semantically secure if the DCRA holds.

2.3 Some Optimizations and Implementation Issues

2.3.1 An Alternative Encryption Function

Let $\psi_s : \mathbb{Z}_{n^s} \times \mathbb{Z}_n^* \rightarrow \mathbb{Z}_{n^{s+1}}^*$ be the isomorphism given by $(x, r) \mapsto (1+n)^x r^{n^s}$ mod n^{s+1} in lemma 2.2. In the above we encrypt an element $m \in \mathbb{Z}_{n^s}$ by a random element from the coset $(\overline{1+n})^m = \psi_s(m, \mathbb{Z}_n^*)$. This element is chosen as $c = \psi_s(m, r)$ for random $r \in \mathbb{Z}_n^*$. Note that if we reduce a ciphertext modulo n, we obtain:

$$c \bmod n = (1+n)^x r^{n^s} \bmod n = r^{n^s} \bmod n$$

The Jacobi symbol modulo n is easy to compute, even without the factors (see e.g. [12]), and since n^s is odd and the Jacobi symbol is multiplicative, we see that from $c = \psi_s(m, r)$, we can compute the Jacobi symbol of r efficiently. Further, by multiplying c by a number of form $\psi_s(0, \tilde{r})$, where $\tilde{r}$ is an arbitrary constant with the same Jacobi symbol as r, we obtain a ciphertext $c' = \psi_s(m, r') = \psi_s(m, r\tilde{r})$, where r' is guaranteed to have Jacobi symbol 1. It easily follows that the cryptosystem is like CS_s, except that we restrict r to have Jacobi symbol 1, is exactly as secure as CS_s under any notion of security. We now exploit this to obtain an alternative and more efficient encryption function.

Using standard techniques we can generate a random RSA modulus $n = pq$ with known p and q such that $p = 3 \bmod 4$, $q = 3 \bmod 4$, $\gcd(p-1, q-1) = 2$. This means that the subgroup of quadratic residues Q_n is cyclic and has odd order, say τ. We can also ensure that all elements in this subgroup - except for a negligible fraction - are generators. This can be done by picking p, q such that all prime factors in $p-1, q-1$ except 2 are sufficiently large. One extreme special case of this is when n is a safe prime product, which is an option we use later for the threshold version of the scheme.

Let $\mathbb{Z}_n^*[+]$ be the elements with Jacobi symbol 1 in $\mathbb{Z}_n^*$. We have that $\mathbb{Z}_n^*[+]$ contains Q_n, has order 2τ and is also cyclic. Finally, $-1 \in \mathbb{Z}_n^*[+] \setminus \mathrm{Q}_n$ by choice of n.

All this implies that if we choose a random $x \in \mathbb{Z}_n^*$ and let $h = -x^2 \bmod n$ then, except with negligible probability, $\langle h \rangle = \mathbb{Z}_n^*[+]$. This then allows us to generate a uniformly random element from $\mathbb{Z}_n^*[+]$ as $h^r \bmod n$, where r is a uniformly random integer from $[0, (p-1)(q-1)/2)$. However, since $(p-1)(q-1)/2$ is the secret key, this would allow only the owner of the secret key to encrypt, which would of course be useless. We can remedy this by using a result from [42]. Let (n, h) be generated as above. Let r be a uniformly random integer from $[0, (p-1)(q-1)/2)$ and let r' be a uniformly random element from $[0, 2^{\lceil k/2 \rceil})$. Then by [42, theorem 3.2] the random variables $(n, h, h^r \bmod n)$ and $(n, h, h^{r'} \bmod n)$ are computationally indistinguishable assuming the intractability of factoring, which is implied by the DCRA. This means that even though $h^{r'} \bmod n$ is not a uniformly random element from $\mathbb{Z}_n^*[+]$, it cannot be distinguished from a uniformly random element from $\mathbb{Z}_n^*[+]$ by any polynomial time algorithm, which suffices for our application. This gives us the following cryptosystem CS'_s.

Key Generation: Choose an admissible RSA modulus $n = pq$ of length k

bits, where $p \equiv q \equiv 3 \pmod 4$, $\gcd(p-1, q-1) = 2$, and such that a random square generates Q_n, except with negligible probability. Choose a generator h of $\mathbb{Z}_n^*[+]$ as described above. Now the public key is $pk = (n, h)$ while the secret key is $\lambda = (p-1)(q-1)/2$, the least common multiple of $(p-1)$ and $(q-1)$.

Encryption: Given a plaintext $m \in \mathbb{Z}_{n^s}$, choose a random $r \in \mathbb{Z}_{2^{\lceil k/2 \rceil}}$ and let the ciphertext be

$$E'_{s,pk}(m, r) = (1+n)^m (h^r \bmod n)^{n^s} \bmod n^{s+1} = E_{s,pk}(m, h^r \bmod n)$$

Decryption: As before.

The following theorem follows directly from the fact that $h^r \bmod n$ is pseudo-random in $\mathbb{Z}_n^*$ under DCRA, that h can be generated given just n, and that the security of CS_s is unchanged when restricting the randomness to Jacobi symbol 1.

Theorem 2.3 *For any s, the cryptosystem CS'_s is semantically secure if and only if the DCRA is true. This holds even if s is allowed to increase polynomially in the security parameter.*

From an exact security point of view, one should be aware that in order to argue the security, we are using the DCRA twice: first to argue that CS'_s is as secure as CS_s (namely $h^r \bmod n$ is pseudo-random) and then to argue that CS_s is secure. This means that if we want to build instances of CS'_s that can be broken with advantage no larger than instances of CS_s with security parameter k, we need to use moduli that are somewhat longer than k bits. How much longer depends on exactly how strong assumptions we are willing to make, and on the complexity of the reduction in the result of [42]. This may partly eliminate the efficiency advantage we show below for CS'_s. On the other hand, this issue can be completely avoided by using more randomness. We can choose r as a random number modulo $n/2$, instead of a random $k/2$-bit number. Then $h^r \bmod n$ will be statistically close to a random element in $\langle h \rangle$ without any assumptions, and up to a negligible term we have the same security for this variant of CS'_s as for CS_s. This will only cost a factor of 2 in performance of the encryption. In the following we will use θ to mean a sufficiently large value, which can be either $2^{k/2} - 1$ or $n/2$ depending on the scenario.

2.3.2 Optimizations of Encryption and Decryption

Encryption

While encrypting, instead of computing $(1+n)^m$ directly, we can compute it according to:

$$(1+n)^m = 1 + mn + \binom{m}{2} n^2 + \ldots + \binom{m}{s} n^s \bmod n^{s+1}$$

this trades an exponentiation with an $\mathcal{O}(n^s)$ size exponent for $\mathcal{O}(s)$ multiplications by calculating the binomials using:

$$\binom{m}{j} = \binom{m}{j-1}\frac{m-j+1}{j} \qquad \text{for } j > 0$$

In the j'th step we calculate $\binom{m}{j}n^j \bmod n^{s+1}$, and since there is a multiplication by n^j it is enough to calculate the binomial modulo n^{s-j+1}. To further optimize the computations the $(j!)^{-1}n^j$ can be precomputed. The pseudo algorithm for calculating the $(n+1)^m$ part of the encryption looks like this (where *precomp* is an array of the precomputed values, $precomp[j] := (j!)^{-1}n^j \bmod n^{s+1}$):

```
c := 1 + mn;
tmp := m;
for j := 2 to s do
begin
  tmp := tmp · (m − j + 1) mod n^(s−j+1);
     /* tmp = m(m − 1) · · · (m − j + 1) mod n^(s−j+1) */
  c := c + tmp · precomp[j] mod n^(s+1);
     /* c = 1 + mn + · · · + (m choose j) n^j mod n^(s+1) */
end
```

In the cryptosystem CS'_s the elements from H are generated as $(h^r \bmod n)^{n^s} \bmod n^{s+1}$ which, if computed naively, certainly leads to no optimization. However, a simple observation allows us to reduce the number of steps used in this computation. Let $h_s = h^{n^s} \bmod n^{s+1}$. Then using the isomorphism from theorem 2.1, we have

$$(h^r \bmod n)^{n^s} \bmod n^{s+1} = \psi_s(0, h^r \bmod n) = \psi_s(0,h)^r = h_s^r \bmod n^{s+1}$$

It follows that $E'_{s,pk}(m,r) = (1+n)^m h_s^r \bmod n^{s+1}$. If we precompute and save the different 2 powers of h_s, then using standard methods for exponentiation with a fixed base, $h_s^r \bmod n^{s+1}$ can be computed by an expected number of $k/4$ multiplications modulo n^{s+1}, and hence the entire encryption can be done in $k/4 + 2s$ multiplications. Compared to a straightforward implementation of CS_s with the same k value, where 2 full scale exponentiations are made, this saves a factor of about $4s$ in computing time, and in particular this is four times as fast as Paillier's original system.

For the length-flexible cryptosystem CS^* from section 2.2.2 this would mean that we would have to compute $h_1, ..., h_{s'}$ for some upper bound s' of the possible values of s. This results in $\mathcal{O}(s'^2 k)$ extra information passed along with the public key, and if someone wants to encrypt a message with $s > s'$ the encryption needs to be performed without the generator. This problem however

can be helped by the following observation:

$$\begin{aligned}
h_{s+1} &= h_s^n \bmod n^{s+2} \\
&= (h^{n^s} \bmod n^{s+1})^n \bmod n^{s+2} \\
&= (h^{n^s} + kn^{s+1})^n \bmod n^{s+2} \quad \text{for some } k \\
&= h^{n^{s+1}} + (h^{n^s})^{n-1}\binom{n}{1}(kn^{s+1}) + (h^{n^s})^{n-2}\binom{n}{2}(kn^{s+1})^2 + \cdots \bmod n^{s+2} \\
&= h^{n^{s+1}} \bmod n^{s+2}
\end{aligned}$$

Computing h_s^n uses $3k/2$ multiplications and to compute h_s from $h_{s'}$ (where $s > s'$) an expected $(s - s')3k/2$ number of multiplication are needed. There are several ways to chose which $h_{s'}$ values to store. First one can simply store all $h_{s'}$ values up to a bound s, in which case we have the value $h_{s'}$ for all s' smaller than s, but this requires $\mathcal{O}(ks^2)$ bits of storage. The other extreme is to store only h_1, but this requires an order of $\mathcal{O}(ks')$ multiplications to compute $h_{s'}$. This however only requires $\mathcal{O}(k)$ bits of storage. We can also make a trade-off where we store the values $h_1, h_3, h_6, ..., h_s$ (the indeces are the points of $\frac{x^2+x}{2}$). There are $\sqrt{s}$ of these values and for any value $s' < s$ the distance to the largest index smaller than or equal to s' is of the order $\mathcal{O}(\sqrt{s'})$. This results in an order of $\mathcal{O}(k\sqrt{s'})$ multiplications to get $h_{s'}$ with $\mathcal{O}(ks\sqrt{s})$ bits of storage.

Decryption

The technique of precomputing factors in binomial coefficients to make encryption faster also applies to the corresponding computations in decryption (see the algorithm in the proof of theorem 2.1). Also in the same way as with encryption, we can exploit the fact that the algorithm involves modular multiplication of a variable by a power of n, which means the value of that variable only needs to be known modulo a smaller power of n.

Another thing that can be optimized is the use of the L function. In the algorithm from theorem 2.1, the L function is calculated once per iteration of the for-loop. Instead of doing this we can calculate the largest of these: $L(a \bmod n^{s+1})$, where $a = (n+1)^m$, and use the property that $L(a \bmod n^{j+1}) = L(a \bmod n^{s+1}) \bmod n^j$. This means that we can remove all but 1 of the divisions and the modular reductions we do are smaller.

The standard trick of splitting up the computations and doing them modulo relatively prime parts of the modulus can also be used here. In the j'th run of the outer loop the moduli will be p^j and q^j. One should be aware that we need to use different L functions for p and q, namely $L_q(a) = ((a - 1 \bmod q^{s+1})/q) \cdot p^{-1} \bmod q^s$ and $L_p(a) = ((a - 1 \bmod p^{s+1})/p) \cdot q^{-1} \bmod p^s$.

In this case, decryption can be sped up by precomputing p^j, q^j for $1 \leq j \leq s$, and $n^{k-1}k!^{-1} \bmod p^j$, $n^{k-1}k!^{-1} \bmod q^j$ for $2 \leq k \leq j \leq s$.

Performance Evaluations

In figures 2.1 and 2.2 the generalized cryptosystem is compared to the El Gamal and RSA cryptosystems. The table is focused on a fixed plaintext size and vari-

	El Gamal (full size)	Generalized Paillier		RSA
		$s=1$	$s=2$	
Security	2048	2048	1024	2048
Ciphertext size	4096	4096	3072	2048
Expansion factor	2	2	1.5	1
Encryption (ms)	1980	1969	578	8
Decryption (ms)	996	1030	312	272

Figure 2.1: Comparison with 2048 bit plaintext size, using a Java implementation.

	El Gamal (full size)	Generalized Paillier				RSA
		$s=1$	$s=2$	$s=3$	$s=4$	
Security	4096	4096	2048	1366	1024	4096
Ciphertext size	8192	8192	6144	5462	5120	4096
Expansion factor	2	2	1.5	1.33	1.25	1
Encryption (ms)	15205	15264	4397	2370	1591	32
Decryption (ms)	7611	7779	2290	1281	873	2001

Figure 2.2: Comparison with 4096 bit plaintext size, using a Java implementation.

able size of security parameter for the generalized cryptosystem. This comparison corresponds to a scenario where you need a certain fixed plaintext size (for instance a large scale election) and it might be sufficient with a smaller security parameter. It shows that if the security parameter is not the same size as the plaintexts encrypted, a significant performance improvement can be achieved.

In figure 2.3 there is a comparison with the number of milliseconds it takes to encrypt a bit using same security parameter, but a variable block size (the amount of plaintext encrypted in a single application of the encryption function). It shows that using El Gamal and the generalized cryptosystem achieves almost the same rates of encryption. It also shows - as expected - that the encryption time per bit increases somewhat with larger s values. Thus, if small ciphertext expansion and large block size is important, this can be achieved at a reasonable performance penalty; but if speed is the only important parameter, $s = 1$ is the best choice.

2.4 Some Building Blocks

2.4.1 A Threshold Variant of the Scheme

What we wish to achieve in this section is a way to distribute the secret key to a set of servers, such that any subset of at least $t+1$ of them can do decryption efficiently, while t or less servers have no useful information. Of course this must be done without degrading the security of the system.

Security parameter	El Gamal	Generalized Paillier $s=1$	$s=2$	$s=4$	RSA
Encryption					
1024	0.264	0.262	0.284	0.387	0.002
2048	0.967	0.955	1.067	1.480	0.004
4096	3.711	3.705	4.146	5.755	0.008
8192	14.467	14.507	16.244	22.617	0.015
Decryption					
1024	0.132	0.149	0.153	0.214	0.039
2048	0.489	0.503	0.559	0.780	0.132
4096	1.865	1.898	2.128	2.958	0.486
8192	7.286	7.349	8.244	11.461	1.854

Figure 2.3: ms per bit encrypted/decrypted on a 750 MHz Pentium III using a java implementation.

In [57] Shoup proposes an efficient threshold variant of RSA signatures. The main part of this is a protocol that allows a set of servers to collectively and efficiently raise an input number to a secret exponent modulo an RSA modulus n. A little more precisely: on input b, each server returns a share of the result, together with a proof of correctness. Given sufficiently many correct shares, these can be efficiently combined to compute $b^d \bmod n$, where d is the secret exponent.

As we explain below it is quite simple to transform this method to our case, thus allowing the servers to raise an input number to our secret exponent d modulo n^{s+1}. So we can solve our problem by first letting the servers help us compute $E_{s,pk}(m,r)^d \bmod n^{s+1}$. Then if we use $g = n+1$ and choose d such that $d = 1 \bmod n^s$ and $d = 0 \bmod \lambda$, the remaining part of the decryption is easy to do without knowledge of d.

We warn the reader that this is only secure for the particular choice of d we have made. For instance, if we had used Paillier's original choice $d = \lambda$, then seeing the value $E_{s,pk}(m,r)^d \bmod n^{s+1}$ would allow an adversary to compute λ and break the system completely. However, in our case, the exponentiation result can safely be made public, since it contains no trace of the secret λ.

A more concrete description: Compared to [57] we still have a secret exponent d, but there is no public exponent e, so we will have to do some things slightly differently toward the end of the decryption process. We will assume that there are w decryption servers, and a threshold of $t < w/2$ so that t servers cannot decrypt, but $t+1$ can decrypt. We will use as modulus n a product of safe primes, i.e. $n = pq$, where $p, q, p' = (p-1)/2, q' = (q-1)/2$ are primes.

We will need as a subroutine a zero-knowledge proof that for given values $u, \tilde{u}, v, \tilde{v} \in \mathbb{Z}^*_{n^{s+1}}$, it holds that $log_u(\tilde{u}) = log_v(\tilde{v})$. Here, it is guaranteed that all values are in the group of squares modulo n^{s+1}, and that v generates the entire group of squares. Note that this group is always cyclic of order $n^s p' q'$, since n is a safe prime product.

A protocol for this can be easily derived from the corresponding one in [57], and works as follows:

Protocol for equality of discrete logs:
Input: $u, \tilde{u}, v, \tilde{v} \in Z^*_{n^{s+1}}$.
Private input for P**:** y such that $y = log_u(\tilde{u}) = log_v(\tilde{v})$ (in our application, the length of y will be at most $(s+1)k$ bits, where k is the modulus length).

1. P chooses a number r of length $(s+1)k+2k_2$ bits at random and sends $a = u^r \bmod n^{s+1}, b = v^r \bmod n^{s+1}$ to the verifier V. Here, k_2 is a secondary security parameter.

2. V chooses a random challenge e of length k_2 bits.

3. P sends to V the number $z = r + ey$.

4. V checks that $u^z = a\tilde{u}^e \bmod n^{s+1}, v^z = b\tilde{v}^e \bmod n^{s+1}$.

This protocol can be made non-interactive using the Fiat-Shamir heuristic and a hash function $\mathcal{H}$: the prover computes a, b as above, sets $e = \mathcal{H}(a, b, u, \tilde{u}, v, \tilde{v})$, computes the reply z as above and defines the proof to be (e, z). To verify such a proof, one checks that $e = \mathcal{H}(u^z\tilde{u}^{-e}, v^z\tilde{v}^{-e}, u, \tilde{u}, v, \tilde{v})$. Assuming the random oracle model, i.e. replacing $\mathcal{H}$ by a random function, one can show soundness and zero-knowledge of this protocol. This is done in exactly the same way as in [57] since, like Shoup, we are working in a cyclic group with only large prime factors in its order. We leave the details to the reader.

This leads to a threshold version CS^t_s of the cryptosystem CS_s:

Key generation: Key generation starts out as in [57]: we find 2 primes p and q that satisfy $p = 2p' + 1$ and $q = 2q' + 1$, where p' and q' are primes and different from p and q. We set $n = pq$ and $\tau = p'q'$. We decide on some $s > 0$, thus the plaintext space will be $\mathbb{Z}_{n^s}$. We calculate a d that satisfies $d = 0 \bmod \tau$ and $d = 1 \bmod n^s$. Now we make the polynomial $f(X) = \sum_{i=0}^{t} a_i X^i \bmod n^s\tau$, by picking a_i (for $0 < i \leq t$) as random values from $\{0, \cdots, n^s * \tau - 1\}$ and $a_0 = d$. The secret share of the i'th authority will be $s_i = f(i)$ for $1 \leq i \leq w$ and the public key will be (n, s). For verification of the actions of the decryption servers, we need the following fixed public values: v, generating the cyclic group of squares in $\mathbb{Z}^*_{n^{s+1}}$ and for each decryption server a verification key $v_i = v^{\Delta s_i} \bmod n^{s+1}$, where $\Delta = w!$.

Encryption: To encrypt a message $m \in \mathbb{Z}_{n^s}$, a random $r \in \mathbb{Z}^*_n$ is picked and the ciphertext is computed as $c = (n+1)^m r^{n^s} \bmod n^{s+1}$. As seen in the previous schemes a generator h can be chosen to improve efficiency. Since this only affects the encryption it will not affect the security of the threshold decryption scheme.

Share decryption: The i'th authority will compute $c_i = c^{2\Delta s_i}$, where c is the ciphertext. Along with this will be a zero-knowledge proof as described above that $log_{c^4}(c_i^2) = log_v(v_i)$, which will convince us that he has indeed raised c to his secret exponent s_i.

Share combining: If we have the required $t+1$ (or more) number of shares with a correct proof, we can combine them into the result by taking a subset S of $t+1$ shares and combine them to

$$c' = \prod_{i \in S} c_i^{2\lambda_{0,i}^S} \bmod n^{s+1} \qquad \text{where } \lambda_{0,i}^S = \Delta \prod_{i' \in S \setminus i} \frac{-i'}{i-i'} \in Z$$

The value of c' will have the form $c' = c^{4\Delta^2 f(0)} = c^{4\Delta^2 d}$. Noting that $4\Delta^2 d = 0 \bmod \lambda$ and $4\Delta^2 d = 4\Delta^2 \bmod n^s$, we can conclude that $c' = (1+n)^{4\Delta^2 m} \bmod n^{s+1}$, where m is the desired plaintext, so this means we can compute m by applying the algorithm from theorem 2.1 and multiplying the result by $(4\Delta^2)^{-1} \bmod n^s$.

Compared to the scheme proposed in [37], there are some technical differences, apart from the fact that [37] only works for the original Paillier version modulo n^2: in [37], an extra random value related to the public element g is part of the public key and is used in the Share combining algorithm. This is avoided in our scheme by the way we chose d, and thus we get a slightly shorter public key and a slightly simpler decryption algorithm.

The system as described requires a trusted party to set up the keys. This may be acceptable as this is a once and for all operation, and the trusted party can delete all secret information as soon as the keys have been distributed. However, using multi-party computation techniques it is also possible to do the key generation without a trusted party. In particular, ideas from [5] can be used to give a reasonably efficient solution. An alternative solution is to use a wider class of moduli by using the result of Damgård and Koprowski [32].

Note that the key generation phase requires that a value of the parameter s is fixed. This means that the system will be able to handle messages encrypted modulo $n^{s'+1}$, for any $s' \leq s$, simply because the exponent d satisfies $d = 1 \bmod n^{s'}$, for any $s' \leq s$. But it will not work if $s' > s$. If a completely general decryption procedure is needed, this can be done as well: if we assume that λ is secret-shared in the key setup phase, the servers can compute a suitable d by running a secure protocol that first inverts λ modulo n^s to get some x as result, and then computes the product $d = x\lambda$ (over the integers). This does not require generic multi-party computation techniques but can be done quite efficiently using techniques from [8]. Note that, while this does require communication between servers, it is not needed for every decryption but only once for every value of s that is used.

We can now show in the random oracle model that this threshold version is as secure as a centralized scheme where one trusted player does the decryption[2].

[2] In fact the random oracle will be needed only to ensure that the non-interactive proofs of correctness of shares will work. Doing these proofs interactively instead would allow us to dispense with the random oracle.

In particular the threshold version is secure relative to the same complexity assumption as the basic scheme. This can be done in a model where a static adversary corrupts up to t players from the start. Concretely, we have:

Theorem 2.4 *Assume the random oracle model and a static adversary that corrupts up to t players from the beginning. Then we have: Given any ciphertext, the decryption protocol outputs the correct plaintext, except with negligible probability. Given an oracle that given a ciphertext returns the corresponding plaintext, the adversary's view of key generation and of the decryption protocol can be efficiently simulated with a statistically indistinguishable distribution.*

The proof follows very closely the corresponding proof in [57]. So here we only sketch the basic ideas: correctness of the scheme is immediate assuming that the adversary can contribute incorrect values for the c_i's with only negligible probability. This, in turn, is ensured by soundness of the zero-knowledge proofs given for each c_i.

For the simulation, we start from the public key n. Then we can simulate the shares $s_{i_1}, ..., s_{i_t}$ of the bad players by choosing them as random numbers modulo n^{s+1}. This will be statistically indistinguishable from the real values, which are chosen modulo $n^s p' q'$. Since d is fixed by the choice of n, this means that the shares of uncorrupted players and the polynomial f are now fixed as well. In particular we have $f(i_1) = s_{i_1}, ..., f(i_t) = s_{i_t}$, but d, f are not easy for the simulator to compute.

However, if we simulate v by choosing it as a ciphertext with known plaintext m_0, i.e. $v = (n+1)^{m_0} r^{2n^s} \bmod n^{s+1}$, we can also compute what $v^{f(0)}$ would be, namely $v^{f(0)} = v^d \bmod n^{s+1} = (1+n)^{m_0} \bmod n^{s+1}$. Let S be the set $0, i_1, ..., i_t$ of $t+1$ indices, and let

$$\lambda^S_{j,i} = \Delta \prod_{i' \in S \setminus i} \frac{j - i'}{i - i'}$$

be the Lagrange coefficients for interpolating the value of a polynomial in point j (times Δ) from its values in points in S. Then we can compute correct values of v_j for uncorrupted players as

$$v_j = \prod_{i \in S} (v^{f(i)})^{\lambda^S_{j,i}}$$

When we get a ciphertext c as input, we ask the oracle for the plaintext m. This allows us to compute $c^d = (1+n)^m \bmod n^{s+1}$. Again this means we can interpolate and compute the contributions c_i from the uncorrupted players. Finally, the zero-knowledge property is invoked to simulate the proofs that these c_i are correct.

2.4.2 Some Auxiliary Protocols

Suppose a prover P presents a skeptical verifier V with a ciphertext c and claims that it encodes plaintext m, or more precisely that he knows r such that $c = E_{s,pk}(m, r)$. A trivial way to convince V would be to reveal also the random

choice r, then V can verify himself that $c = E_{s,pk}(m,r) = (1+n)^m r^{n^s} \bmod n^{s+1}$. However, for use in the following, we need a solution where no extra useful information is revealed.

It is easy to see that this is equivalent to convincing V, that $c(1+n)^{-m} \bmod n^{s+1}$ is an encryption of 0, or equivalently that it is an n^s'th power. So we now propose a protocol for this purpose which is a simple generalization of the one from [44].

We note that this and the following protocols are not zero-knowledge as they stand, only honest verifier zero-knowledge. However, first zero-knowledge protocols for the same problems can be constructed from them using standard methods, and secondly, in our applications we will always be using them in a non-interactive variant based on the Fiat-Shamir heuristic, which means that we cannot obtain zero-knowledge. However, we can obtain security in the random oracle model. As for soundness, we prove that the protocols satisfy so-called special soundness (see [20]), which in particular implies that they satisfy standard knowledge soundness.

Protocol for n^s'th powers
Input: n, g, u.
Private input for P: $v \in \mathbb{Z}_n^*$, such that $u = E_{s,pk}(0,v)$.

1. P chooses r at random in $\mathbb{Z}_n^*$ and sends $a = E_{s,pk}(0,r)$ to V.

2. V chooses e, a random k_2 bit number, and sends e to P.

3. P sends $z = rv^e \bmod n$ to V.

4. V checks that u, a, z are prime to n and that $E_{s,pk}(0,z) = au^e \bmod n^{s+1}$, and accepts if and only if this is the case.

It is now simple to show:

Lemma 2.3 *The above protocol is complete honest verifier zero-knowledge, and satisfies that from any pair of accepting conversations (between V and any prover) of form $(a,e,z),(a,e',z')$ with $e \neq e'$, one can efficiently compute a v such that $u = E_{s,pk}(0,v)$, provided 2^{k_2} is smaller than the smallest prime factor of n.*

Proof. For completeness, we just plug into the equation that V checks, by lemma 2.2 we get

$$\begin{aligned} au^e &= E_{s,pk}(0,r)E_{s,pk}(0,v)^e = E_{s,pk}(0, rv^e \bmod n) \\ &= E_{s,pk}(0,z) \bmod n^{s+1} \end{aligned}$$

For honest verifier simulation, the simulator chooses a random $z \in \mathbb{Z}_n^*$, a random e, sets $a = E_{s,pk}(0,z)u^{-e} \bmod n^{s+1}$ and outputs (a,e,z). This is easily seen to be a perfect simulation.

For the last claim, observe that since the conversations are accepting, we have $E_{s,pk}(0, z) = au^e \bmod n^{s+1}$ and $E_{s,pk}(0, z') = au^{e'} \bmod n^{s+1}$, so we get

$$E_{s,pk}(0, z/z' \bmod n) = u^{e-e'} \bmod n^{s+1}$$

Since $e - e'$ is prime to n by the assumption on 2^{k_2}, choose α, β such that $\alpha n^s + \beta(e - e') = 1$. Let $\bar{u} = u \bmod n$ and set $v = \bar{u}^\alpha (z/z')^\beta \bmod n$. Notice that $u^{n^s} \bmod n^{s+1} = E_{s,pk}(0, u \bmod n) = E_{s,pk}(0, \bar{u})$. We then get

$$E_{s,pk}(0, v) = E_{s,pk}(0, \bar{u})^\alpha E_{s,pk}(0, z/z')^\beta = u^{\alpha n^s} u^{\beta(e-e')} = u \bmod n^{s+1}$$

so that v is indeed the desired n^s'th root of u. □

In our application of this protocol, the modulus n will be chosen by a trusted party or by a multi-party computation, such that n has two prime factors of roughly the same size. Hence, if k is the bit length of n, we can set $k_2 = k/2$ or $k_2 = 160$ and be assured that a cheating prover can make the verifier accept with probability $\leq 2^{-k_2}$.

The lemma immediately implies, using the techniques from [20], that we can build an efficient proof that an encryption contains one of two given values, without revealing which one it is. Given the encryption C and the two candidate plaintexts m_1, m_2, prover and verifier compute $u_1 = C/g^{m_1} \bmod n^{s+1}, u_2 = C/g^{m_2} \bmod n^{s+1}$, and the prover shows that either u_1 or u_2 encrypt 0 and also proves knowledge of one of the corresponding n^s'th roots. This can be done using the following protocol, where we assume without loss of generality that the prover knows v_1 such that $u_1 = E_{s,pk}(0, v_1)$, and where M denotes the honest-verifier simulator for the n^s-power protocol above:

Protocol 1-out-of-2 n^s'th power
Input: n, g, u_1, u_2.
Private input for P: v_1, such that $u_1 = E_{s,pk}(0, v_1)$.

1. P chooses r_1 at random in $\mathbb{Z}_n^*$. He invokes M on input n, u_2 to get a conversation a_2, e_2, z_2. He sends $a_1 = E_{s,pk}(0, r_1), a_2$ to V.

2. V chooses e, a random k_2 bit number, and sends e to P.

3. P computes $e_1 = e - e_2 \bmod 2^{k_2}$ and $z_1 = r_1 v_1^{e_1} \bmod n$. He then sends e_1, z_1, e_2, z_2 to V.

4. V checks the equations $e = e_1 + e_2 \bmod 2^{k_2}$, $E_{s,pk}(0, z_1) = a_1 u_1^{e_1} \bmod n^{s+1}$, $E_{s,pk}(0, z_1) = a_2 u_2^{e_2} \bmod n^{s+1}$, and that $u_1, u_2, a_1, a_2, z_1, z_2$ are relatively prime to n. He accepts if and only if this is the case.

The proof techniques from [20] and lemma 2.3 immediately imply

Lemma 2.4 *Protocol* 1-out-of-2 n^s'th power *is complete, honest verifier zero-knowledge, and satisfies that from any pair of accepting conversations (between V and any prover) of form $(a_1, a_2, e, e_1, z_1, e_2, z_2)$, $(a_1, a_2, e', e_1', z_1', e_2', z_2')$ with $e \neq e'$, one can efficiently compute v, such that either $u_1 = E_{s,pk}(0, v)$ or $u_2 = E_{s,pk}(0, v)$, provided 2^{k_2} is less than the smallest prime factor of n.*

Our final building block allows a prover to convince a verifier that three encryptions contain values a, b and c such that $ab = c \bmod n^s$. For this, we propose a protocol inspired by a similar construction found in [18].

Protocol Multiplication-mod-n^s
Input: n, g, e_a, e_b, e_c.
Private input for P: values a, b, c, r_a, r_b, r_c such that $ab = c \bmod n$ and $e_a = E_{s,pk}(a, r_a)$, $e_b = E_{s,pk}(b, r_b)$, $e_c = E_{s,pk}(c, r_c)$.

1. P chooses random values $d \in \mathbb{Z}_{n^s}$, $r_d, r_{db} \in \mathbb{Z}_n^*$ and sends to V encryptions $e_d = E_{s,pk}(d, r_d), e_{db} = E_{s,pk}(db, r_{db})$.

2. V chooses e, a random k_2-bit number, and sends it to P.

3. P opens the encryption $e_a^e e_d = E_{s,pk}(ea + d, r_a^e r_d \bmod n)$ by sending $f = ea + d \bmod n^s$ and $z_1 = r_a^e r_d \bmod n$. Finally, P also opens the encryption $e_b^f (e_{db} e_c^e)^{-1} = E_{s,pk}(0, r_b^f (r_{db} r_c^e)^{-1} \bmod n)$ by sending $z_2 = r_b^f (r_{db} r_c^e)^{-1} \bmod n$.

4. V verifies that the openings of encryptions in the previous step were correct, that all values sent by P are relatively prime to n, and accepts if and only if this was the case.

Lemma 2.5 *Protocol* Multiplication-mod-n^s *is complete, honest verifier zero-knowledge, and satisfies that from any pair of accepting conversations (between V and any prover) of form $(e_d, e_{db}, e, f, z_1, z_2)$, $(e_d, e_{db}, e', f', z_1', z_2')$ with $e \neq e'$, one can efficiently compute the plaintext a, b, c corresponding to e_a, e_b, e_c such that $ab = c \bmod n^s$, providing 2^{k_2} is smaller than the smallest prime factor in n.*

Proof. Completeness is clear by inspection of the protocol. For honest verifier zero-knowledge, observe that the equations checked by V are $e_a^e e_d = E_{s,pk}(f, z_1) \bmod n^{s+1}$ and $e_b^f (e_{db} e_c^e)^{-1} = E_{s,pk}(0, z_2) \bmod n^{s+1}$. From this it is clear that we can generate a conversation by choosing first f, z_1, z_2, e at random, and then computing e_d, e_{db} that will satisfy the equations. This only requires inversion modulo n^{s+1}, and generates the right distribution because the values f, z_1, z_2, e are also independent and random in the real conversation. For the last claim, note first that since encryptions uniquely determine plaintexts, there are fixed values a, b, c, d contained in e_a, e_b, e_c, e_d, and a value x contained in e_{db}. The fact that the conversations given are accepting implies that $f = ea + d \bmod n^s$, $f' = e'a + d \bmod n^s$, $fb - x - ec = 0 = f'b - x - e'c \bmod n^s$. Putting this together, we obtain $(f - f')b = (e - e')c \bmod n^s$ or $(e - e')ab = (e - e')c \bmod n^s$. Now, since $(e - e')$ is invertible modulo n^s by assumption on 2^{k_2}, we can conclude that $c = ab \mod n^s$ (and also compute a, b and c). □

The protocols from this section can be made non-interactive using the standard Fiat-Shamir heuristic of computing the challenge from the first message using a hash function. This can be proved secure in the random oracle model.

Furthermore, although the protocols here have been phrased so they can be used to prove statements on values encrypted in CS_s, they can also be directly used in the same way for values encrypted under the more efficient variant CS'_s. This follows from the fact that, if for a given $u \in \mathbb{Z}^*_{n^{s+1}}$ you know $m, \tilde{v}$ such that $u = E'_{s,pk}(m, \tilde{v})$, we have $u = E_{s,pk}(m, h^{\tilde{v}} \bmod n)$. In other words you can efficiently compute v such that $u = E_{s,pk}(m, v)$. Thus a prover can use u in any of the above protocols pretending it was encrypted using CS_s. Note that this applies to both ciphertexts that are input to the protocols, and those that are generated by the prover during executions.

2.5 Introducing an El Gamal Element

In section 2.4.1, a threshold decryption for a fixed s, and a sketch for making it length-flexible (i.e. work for encryptions using different values of s) were introduced. However, this requires an expensive multi-party computation each time a new s is used (assuming the old values are stored). In the following sections we will add an El Gamal element [35]. This allows us to make a more efficient threshold decryption, at the cost of an extra assumption:

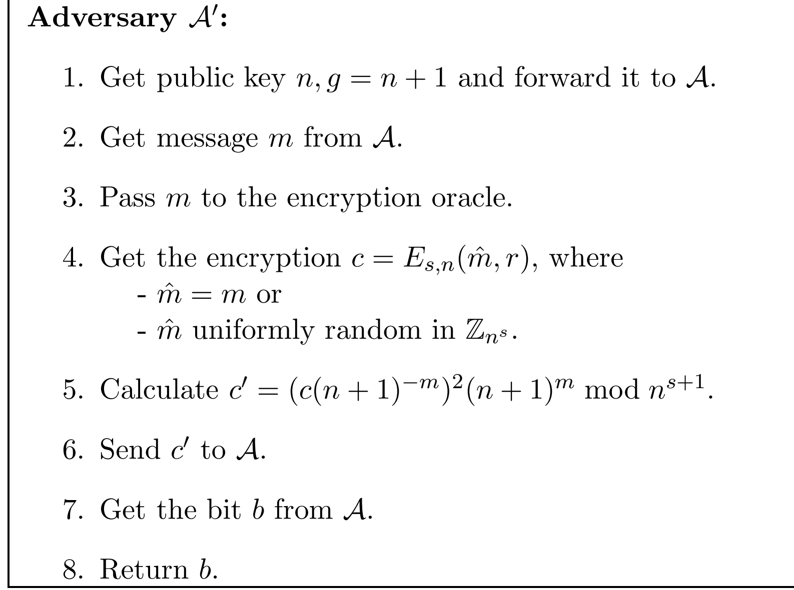
Conjecture 2.2 (Decisional Diffie-Hellman (composite DDH)) *Let $\mathcal{A}$ be any probabilistic polynomial time algorithm, and assume $\mathcal{A}$ gets $(n, g, g^a \bmod n, g^b \bmod n, y)$ as input. Here $n = pq$ is an admissible RSA modulus of length k bits, g is a element of Q_n, the group of squares in $\mathbb{Z}^*_n$. The values a and b are chosen uniformly at random in $\mathbb{Z}_{\varphi(n)/4}$, and the value y is either random in Q_n or satisfies $y = g^{ab} \bmod n$. $\mathcal{A}$ outputs a bit b. Let $p_0(\mathcal{A}, k)$ be the probability that $b = 1$ if y is random in Q_n, and $p_1(\mathcal{A}, k)$ the probability that $b = 1$ if $y = g^{ab} \bmod n$. Then $|p_0(\mathcal{A}, k) - p_1(\mathcal{A}, k)|$ is negligible in k.*

Before moving to the threshold system, we will look at what happens when we combine the ideas from section 2.3.1 of using a generator to create the random values with the ideas from [35]. This leads to a new cryptosystem $\overline{CS}$:

Key Generation: Choose an RSA modulus $n = pq$ of length k bits with $p = 2p' + 1$ and $q = 2q' + 1$, where p, q, p', q' are primes. Select a generator $g \in \mathrm{Q}_n$, the group of all squares of $\mathbb{Z}^*_n$, and $\alpha \in \mathbb{Z}_\tau$, where $\tau = p'q' = |\mathrm{Q}_n|$. The public key is then $pk = (n, g, h)$ with $h = g^\alpha \bmod n$ and the private key is α.

Encryption: Given a plaintext $m \in \mathbb{Z}^+$, choose an integer $s > 0$ such that $m \in \mathbb{Z}_{n^s}$ and a random $r \in \mathbb{Z}_\theta$ (here θ is the value introduced on page 18), and let the ciphertext be

$$\begin{aligned}\overline{E}_{pk}(m, r) &= \overline{E}_{(s),pk}(m, r) \\ &= (g^r \bmod n, (h^r \bmod n)^{n^s} (n+1)^m \bmod n^{s+1}) \\ &= (g^r \bmod n, E^*_{(s),n}(m, h^r \bmod n))\end{aligned}$$



Adversary $\mathcal{A}'$:

1. Get public key $n, g = n + 1$ and forward it to $\mathcal{A}$.
2. Get message m from $\mathcal{A}$.
3. Pass m to the encryption oracle.
4. Get the encryption $c = E_{s,n}(\hat{m}, r)$, where
   - $\hat{m} = m$ or
   - $\hat{m}$ uniformly random in $\mathbb{Z}_{n^s}$.
5. Calculate $c' = (c(n+1)^{-m})^2(n+1)^m \bmod n^{s+1}$.
6. Send c' to $\mathcal{A}$.
7. Get the bit b from $\mathcal{A}$.
8. Return b.

Figure 2.4: Algorithm for adversary $\mathcal{A}'$ breaking the cryptosystem CS_s from 2.2 using the adversary $\mathcal{A}$.

Decryption: Given a ciphertext $c = (G, H) = \overline{E}_{pk}(m, r)$, s can be deduced from the length of c (or it can be attached to the encryption) and m can be found as (here $\mathrm{dLog}_s()$ is the function defined on page 11)

$$\begin{aligned} m &= \mathrm{dLog}_s(H(G^\alpha \bmod n)^{-n^s}) \\ &= \mathrm{dLog}_s((g^{\alpha r} \bmod n)^{n^s}(n+1)^m(g^{r\alpha} \bmod n)^{-n^s}) \\ &= \mathrm{dLog}_s((n+1)^m \bmod n^{s+1}) = m \bmod n^s \end{aligned}$$

Remark 2.1 *The key generation above assumes that one knows τ and hence the factorization when choosing α. However, one can also choose α at random in $\mathbb{Z}_\theta$, that is, generate $\alpha, h = g^\alpha$ from n, g only. This makes no difference to security, since the h produced will have an indistinguishable distribution, and it allows n, g to be system constants used by all users.*

2.5.1 Security of the Cryptosystem

Before proving the security of the cryptosystem we prove a lemma stating that the use of Q_n does not degrade the security of the Damgård-Jurik cryptosystem.

Lemma 2.6 *The cryptosystem CS_s, where r is chosen in Q_n, is as semantically secure, with respect to definition 2.1, as the cryptosystem CS_s where $r \in \mathbb{Z}_n^*$.*

Proof. To show that the security is equivalent, assume an adversary $\mathcal{A}$ exists, that can break the semantic security of the quadratic cryptosystem. Then the adversary $\mathcal{A}'$ shown in figure 2.4 breaks the original cryptosystem.

Given an encryption of the message m, the ciphertext c' generated by $\mathcal{A}'$ will be a legal encryption of m with uniform distribution of the random r, which follows from:

$$\begin{aligned}(c(n+1)^{-m})^2(n+1)^m &= (r^{n^s}(n+1)^{m-m})^2(n+1)^m \\ &= (r^2)^{n^s}(n+1)^m \\ &= E_{s,pk}(m, r^2) \bmod n^{s+1}\end{aligned}$$

In the case where $\hat{m}$ is chosen uniformly random from the message space $\mathbb{Z}_{n^s}$, the resulting c' is

$$\begin{aligned}(c(n+1)^{-m})^2(n+1)^m &= (r^{n^s}(n+1)^{\hat{m}-m})^2(n+1)^m \\ &= r^{2n^s}(n+1)^{2\hat{m}-m} \\ &= E_{s,pk}(2\hat{m}-m, r^2) \bmod n^{s+1}\end{aligned}$$

Because $\gcd(2, n) = 1$, the function $2\hat{m} - m$ is a permutation of $\mathbb{Z}_{n^s}$. So the new encryption will be uniformly random in $\mathbb{Z}_{n^s}$, since $\hat{m}$ is. The probabilities for $\mathcal{A}'$ are $p_0(\mathcal{A}', k) = p_0(\mathcal{A}, k)$ and $p_1(\mathcal{A}', k) = p_1(\mathcal{A}, k)$, which means that the advantage of $\mathcal{A}'$ is the same as for $\mathcal{A}$. □

Given the lemma it is easy to prove that the cryptosystem $\overline{CS}$ is semantically secure.

Theorem 2.5 (Semantic Security) *Under the conjectures 2.1 (DCRA) and 2.2 (composite-DDH) the cryptosystem $\overline{CS}$ is semantically secure with respect to definition 2.1.*

Proof. The proof is done in a 3 step hybrid reduction using the composite-DDH conjecture and the semantic security of the cryptosystem defined in lemma 2.6 (which is secure under the DCRA). Given the public key (n, g, h), the following 4 pairs are indistinguishable under conjecture 2.1 and 2.2:

1. $(g^k \bmod n, (h^k)^{n^s}(n+1)^m \bmod n^{s+1})$
2. $(g^k \bmod n, (r)^{n^s}(n+1)^m \bmod n^{s+1})$, where r is uniformly random in Q_n
3. $(g^k \bmod n, (r)^{n^s}(n+1)^{m'} \bmod n^{s+1})$, where m' is random in $\mathbb{Z}_{n^s}$
4. $(g^k \bmod n, (h^k)^{n^s}(n+1)^{m'} \bmod n^{s+1})$

If tuple 1 and 4 are indistinguishable then the $\overline{CS}$ system is semantically secure following the definition. If an adversary can distinguish between tuple 1 and 4 with advantage $\epsilon > \frac{1}{f(k)}$ for some polynomial $f(k)$, there will be an adversary able to distinguish between a pair of consecutive tuples with a probability larger than $\epsilon' > \frac{1}{3f(k)}$. Each of the 3 pairs are shown below to be indistinguishable, thereby showing that an adversary able to distinguish between the tuples 1 and 4 cannot exist.

The pairs 1 and 2 are indistinguishable due to a reduction to composite-DDH. Assuming an adversary $\mathcal{B}$ having a non-negligible advantage of distinguishing pairs 1 and 2, an adversary $\mathcal{B}'$ can be built that will break composite-DDH with the same advantage. The algorithm for the adversary $\mathcal{B}'$ can be seen in figure 2.5.

Adversary $\mathcal{B}'$:

1. Get the composite-DDH tuple: (n, g, g^a, g^b, y).
2. Give the public key $(n, g, h := g^a)$ to $\mathcal{B}$.
3. Get message m from $\mathcal{B}$.
4. Give the encryption $(g^b, y^{n^s}(n+1)^m \bmod n^{s+1})$ to $\mathcal{B}$.
5. Get the bit b from $\mathcal{B}$.
6. Return b.

Figure 2.5: Algorithm for adversary $\mathcal{B}'$ that breaks the composite-DDH given the adversary $\mathcal{B}$.

The 2 cases of y in the composite-DDH correspond to tuple 1 and 2 respectively:

$\mathbf{y = g^{ab}}$: Here $y = g^{ab} = (g^a)^b = h^b$, which is the value used in tuple 1.

y uniformly random in $\mathbf{Q_n}$: This is a random value as in tuple 2.

Thereby tuple 1 and 2 are indistinguishable under the composite-DDH assumption.

That pair 2 and 3 are indistinguishable follow directly from lemma 2.6. This follows since using $r \in \mathrm{Q}_n$ the cryptosystem CS_s is still semantically secure, which implies that the encryption $r^{n^s}(n+1)^m \bmod n^{s+1}$ of a message m is indistinguishable from the encryption of a random message $m' \in \mathbb{Z}_{n^s}$: $r^{n^s}(n+1)^{m'} \bmod n^{s+1}$.

The pairs 3 and 4 are indistinguishable following the same reduction as in the step from 1 to 2. □

This cryptosystem may seem to be simply a combination of El Gamal and Paillier, and hence its security is inherently based on both the above conjectures, but this is in fact not true. A simple modification makes it possible to show semantic security based *only* on conjecture 2.1, using a technique from [23]. Given the same public key (n, g, h), we can set $g' = g^{n^s} \bmod n^{s+1}$ (so that g' generates the subgroup of $\mathbb{Z}^*_{n^{s+1}}$ of order τ), and $h' = h^{n^s} \bmod n^{s+1}$. Encryption is $\tilde{E}_{(s),pk}(m, r) = (g'^r \bmod n^{s+1}, h'^r(n+1)^m \bmod n^{s+1})$.

Theorem 2.6 (Semantic Security of modified system) *Under conjecture 2.1 (DCRA) the modified cryptosystem is semantically secure with respect to definition 2.1.*

Proof. This was proved for $s = 1$ in [23], and follows in essentially the same way for the general system: a ciphertext (G, H) is always of form $(G, G^\alpha(n+1)^m \bmod n^{s+1})$. Now, conjecture 2.1 implies that one cannot distinguish the case where $G \in \langle g' \rangle$ from the case where it is chosen randomly in $\mathbb{Z}^*_{n^{s+1}}[+]$ (the

subgroup of $\mathbb{Z}^*_{n^{s+1}}$ with elements that have Jacobi symbol 1). In the latter case, however, one can verify that if α is chosen large enough, the ciphertext contains no Shannon information on the plaintext. □

Since our basic system is more efficient, we describe our protocols in the following in terms of the $\overline{E}_{(s),pk}(,)$ encryption function, but they can all be modified to use the $\tilde{E}_{(s),pk}(,)$ encryption function instead.

2.6 An Efficient Length-Flexible Threshold Cryptosystem

From the basic cryptosystem, a length-flexible threshold cryptosystem $\overline{CS}^t$ can be constructed by using a threshold exponentiation based on Shoup's threshold signatures [57].

Key Generation: Choose an RSA modulus $n = pq$ and a $g \in \mathrm{Q}_n$ as above. Pick the secret value $a_0 = \alpha \in \mathbb{Z}_\tau$ and some random coefficients $a_i \in \mathbb{Z}_\tau$ for $1 \leq i \leq t$, where $t < w/2$ is the threshold of the system with w servers. The polynomial $f(x) = \sum_{0 \leq i \leq t} a_i x^i$ is created and the secret shares are calculated as $\alpha_i = f(i)$. The public value is $h = g^\alpha \bmod n$, and the values for verification are $h_i = g^{\alpha_i} \bmod n$. The public key is $pk = (n, g, h)$, the verification values $(h_1, ..., h_w)$, and the private key of server i is α_i.

Encryption: Given a plaintext $m \in \mathbb{Z}^+$, choose an integer $s > 0$, such that $m \in \mathbb{Z}_{n^s}$, and pick a random $r \in \mathbb{Z}_\theta$. The ciphertext is then

$$\begin{aligned}\overline{E}^t_{pk}(m, r) &= \overline{E}^t_{(s),pk}(m, r)\\ &= (g^r \bmod n, (h^{4\Delta^2 r} \bmod n)^{n^s}(n+1)^m \bmod n^{s+1})\end{aligned}$$

Threshold Decryption: Given a ciphertext $c = (G, H) = \overline{E}^t_{(s),pk}(m, r)$ each of the servers releases the value:

$$d_i = G^{2\Delta\alpha_i} \bmod n$$

and a proof that $\log_g(h_i) = \log_{G^{4\Delta}}(d_i^2)$. The proof used for this is shown in section 2.7.2. The d_i values, from the set S of servers with valid proofs, are combined using Lagrange interpolation to create the exponent $4\Delta^2\alpha$:

$$d = \prod d_i^{2\lambda_i^S} = G^{4\Delta^2\alpha} = h^{4\Delta^2 r} \bmod n \quad \text{where } \lambda_i^S = \Delta \prod_{j \in S \setminus \{i\}} \frac{-j}{i-j}$$

The reason for the factor Δ is to ensure $\lambda_i^S \in \mathbb{Z}$. Now the $h^{4\Delta^2 r}$ can be removed by calculating

$$H' = Hd^{-n^s} = (n+1)^m \bmod n^{s+1}$$

and the plaintext found as $m = \mathrm{dLog}_s(H' \bmod n^{s+1})$.

2.7 A Proof Friendly Variant

The system from the previous section only works as long as legal encryptions are submitted, so we would like a protocol allowing us to prove in zero-knowledge, that a ciphertext is well formed. A standard problem with building an efficient protocol of this type is, that we need all group elements used to have only large prime factors in their orders. In our case, this can be ensured by squaring them before we start the proofs, i.e. instead of trying to show that g has some desired property, we prove that g^2 has it. However, as we shall see, this only implies that g or $-g$ has the desired property. To handle this, we define a slightly relaxed cryptosystem $\widehat{CS}^t$:

Key Generation: As above.

Encryption: Given a plaintext $m \in \mathbb{Z}^+$, choose an integer $s > 0$ such that $m \in \mathbb{Z}_{n^s}$, and pick a random $r \in \mathbb{Z}_\theta$ and $b_0, b_1 \in \{0, 1\}$. The ciphertext is then

$$\widehat{E}^t_{pk}(m, r, b_0, b_1) = ((-1)^{b_0} g^r \bmod n, \\ (-1)^{b_1}(h^{4\Delta^2 r} \bmod n)^{n^s}(n+1)^m \bmod n^{s+1})$$

Threshold Decryption: Given a ciphertext $c = (G, H) = \widehat{E}^t_{(s),pk}(m, r, b_0, b_1)$, it is only decrypted if the Jacobi symbol of G and H is 1. Since G is squared, the d value can be computed as above. To decrypt, H needs to be squared, so a slightly different computation is made:

$$H' = H^2 d^{-2n^s} = (n+1)^{2m} \bmod n^{s+1}$$

and the plaintext is found as $m = \mathrm{dLog}_s(H')/2 \bmod n^s$.

Proving properties is now easier, since values can be squared to make sure they are in $\mathbb{Q}_n$ during the proof. In section 2.7.2 three proofs are shown: 1) a proof that something is a legal encryption, 2) a proof that something is a legal encryption of some publicly known plaintext, and 3) the threshold decryption proof. Note that the techniques from section 2.3.2 can be used to improve the complexity of most computations from $\mathcal{O}(s^3k^3)$ to $\mathcal{O}(s^2k^3)$.

2.7.1 Security of the Threshold Cryptosystems

Due to its homomorphic properties, our basic cryptosystem cannot be chosen ciphertext secure, so we cannot hope to prove that the threshold version is chosen ciphertext secure either. However, we can show a result saying essentially, that as long as the adversary does not control the ciphertexts being decrypted, the threshold decryption releases no information other than the plaintext.

Definition 2.2 *A* chosen plaintext threshold adversary $\mathcal{A}$ *runs in probabilistic polynomial time and can statically and actively corrupt* $t < w/2$ *of the servers. In addition, for any efficiently samplable distribution* $\mathcal{D}$*, he may request that*

a message m be chosen according to $\mathcal{D}$, and then to see a random ciphertext containing m be decrypted using the threshold decryption protocol. A threshold public-key cryptosystem is secure against such an adversary, if his view can be simulated in probabilistic polynomial time given only the public key.

Note that since $\mathcal{D}$ is arbitrary, this includes the case where the adversary chooses m himself. This type of security will be sufficient in the mix-net in chapter 3. Using a more elaborate decryption protocol, where a ciphertext is randomized before it is decrypted, an even stronger property can be shown, namely that security of the threshold system is equivalent to security of the non-threshold version under any attack.

Lemma 2.7 *The threshold cryptosystems are semantically secure under Conjectures 2.1 and 2.2. They are also secure against any chosen plaintext threshold adversary as defined above.*

Proof. The semantic security follows from theorem 2.5, since an encryption of m in the basic cryptosystem can be transformed into an encryption of $4\Delta^2 m \bmod n^s$ in the threshold systems by raising the last component to the $4\Delta^2$'th power, and in the last system by multiplying with $(-1)^{b_0}$ and $(-1)^{b_1}$.

The proofs of correctness used for the decryption shares are identical to the Shoup verification proofs for signature shares in [57], where they were proved sound and statistical zero-knowledge. Furthermore, the secret sharing of the secret key is identical to the one used in [57]. Hence, it is straightforward to construct a simulation proof of security along the lines of the proof in [57]. So here we only sketch the ideas: given the public key, we can simulate the shares of corrupt players by choosing random values. We can then reconstruct the verification values of honest servers using Lagrange interpolation "in the exponent". To simulate the adversary's view of the decryption protocol, we choose m according to $\mathcal{D}$ and construct a random ciphertext containing m. In particular, this means we know the relevant value of h^r. Given this and the shares of corrupt players, we can compute, with a statistically close distribution, the contribution from honest servers to the decryption. Their proofs of correctness can be simulated. By soundness of the proofs, the adversary will not be able to contribute bad values, so the decryption will output m as desired. □

2.7.2 Proofs in the Proof Friendly Variant

Here we show some protocols for proving various claims on encryptions in the proof friendly cryptosystem. In this system we are working with elements of the form g^r, which pose a problem in the zero-knowledge protocols when we need to find r. So to get soundness in the zero-knowledge protocols we need the Strong RSA assumption:

Conjecture 2.3 *Let $\mathcal{A}$ be any probabilistic polynomial time algorithm, and assume $\mathcal{A}$ gets n, z as input. Here n is a k bit RSA modulus, and z is a random element in $\mathbb{Z}_n^*$. $\mathcal{A}$ outputs two values $y \in \mathbb{Z}_n^*, r > 1$ such that $y^r = z \bmod n$ or*

failure. Let $p(\mathcal{A},k)$ be the probability that $\mathcal{A}$ outputs such a y,r. Then $p(\mathcal{A},k)$ is negligible in k.

Remark 2.2 *Note that this poses a problem with the setup of the system if g is generated by choosing a random element $x \in \mathbb{Z}_n^*$ and setting $g = x^2$. This gives a non-trivial square root of g, which might be used to break the above conjecture. However, this can be fixed by choosing x at random in $\mathbb{Z}_n^*$ with the constraint that it must have Jacobi symbol -1. Because all values in the protocol have Jacobi symbol 1, the value x cannot be used to create a non-trivial square root in these. Furthermore if someone creates a square root of g with Jacobi symbol 1, this together with x can be used to find the factorization of n.*

Proof of Legal Encryption

We prove that given an encryption (G,H) there exist an $r \in \mathbb{Z}_\theta$ and an $m \in \mathbb{Z}_{n^s}$, such that $G = \pm g^r \bmod n$ and $H = \pm(h^{4\Delta^2 r})^{n^s}(n+1)^m$.

Protocol for legal encryption
Input: $n, g, h, c = (G,H)$.
Private input for P**:** $r \in \mathbb{Z}_\theta$ and $m \in \mathbb{Z}_{n^s}$, such that $c = \widehat{E}^t_{(s),pk}(m, r, b_0, b_1)$ for some b_0 and b_1.

1. P chooses r' in $\{0, ..., 2^{|\theta|+2k_2}-1\}$, $b'_0, b'_1 \in \{0,1\}$ and $m' \in \mathbb{Z}_{n^s}$ at random, where k_2 is a secondary security parameter (e.g. 160 bits). P sends $c' = (G', H') = \widehat{E}^t_{(s),pk}(m', r', b'_0, b'_1)$ to V.
2. V chooses e, a random k_2 bit number, and sends e to P.
3. P sends $\hat{r} = r' + er$ and $\hat{m} = m' + em \bmod n^s$ to V.
4. V checks that G, H, G', H' are prime to n, have Jacobi symbol 1 and that the equation $\widehat{E}^t_{(s),pk}(2\hat{m}, 2\hat{r}, 0, 0) = (G'^2G^{2e} \bmod n, H'^2H^{2e} \bmod n^{s+1}) = c'^2c^{2e}$ holds. V accepts if and only if this is the case.

The protocol above can be proven to be sound and complete honest verifier zero-knowledge. This is enough for the election protocol in section 4.4, since it will only be used in an non-interactive setting using the Fiat-Shamir Heuristic and hash function $\mathcal{H}$ to generate the challenge $e = \mathcal{H}(G, H, G', H')$.

The lemma below uses the Strong RSA assumption to ensure that we can find m and r. However, if we just need that it is a legal encryption and not that the prover knows m and r, we do not need the Strong RSA assumption if 2^{k_2} is less than the smallest prime factor of τ. This follows from the fact that $e_1 - e_2$ is invertible modulo τ, so the value r can be computed uniquely modulo τ given knowledge of τ.

Lemma 2.8 *Protocol for* legal encryption *is complete, statistical honest verifier zero-knowledge, and under the Strong RSA assumption (conjecture 2.3) it satisfies that from any pair of accepting conversations (between V and any prover) of form $(c', e_1, \hat{r}_1, \hat{m}_1)$, $(c', e_2, \hat{r}_2, \hat{m}_2)$ with $e_1 \neq e_2$, one can efficiently*

compute m, r*, such that* $c = (G, H) = \widehat{E}^t_{(s),pk}(m, r, b_0, b_1)$ *(for some* b_0, b_1*), provided* 2^{k_2} *is less than the smallest prime factor of* n*.*

Proof. For completeness we can verify the formula used in the protocol:

$$\begin{aligned} c'^2 c^{2e} &= \widehat{E}^t_{(s),pk}(m', r', b'_0, b'_1)^2 \widehat{E}^t_{(s),pk}(m, r, b_0, b_1)^{2e} \\ &= \widehat{E}^t_{(s),pk}(2m', 2r', 0, 0)\widehat{E}^t_{(s),pk}(2em, 2er, 0, 0) \\ &= \widehat{E}^t_{(s),pk}(2(m' + em), 2(r' + er), 0, 0) \\ &= \widehat{E}^t_{(s),pk}(2\hat{m}, 2\hat{r}, 0, 0) \end{aligned}$$

For honest verifier simulation pick $\hat{r} \in \{0, ..., 2^{|\theta|+2k_2} - 1\}$, $\hat{m} \in \mathbb{Z}_{n^s}$, $b_0, b_1 \in \{0, 1\}$ and $e \in \{0, ..., 2^{k_2} - 1\}$. Set $c' = \widehat{E}^t_{(s),pk}(\hat{m}, \hat{r}, b_0, b_1)c^{-e}$. This can easily be seen to be a statistically close simulation.

To prove the claim about recovering m and r we look at the 2 accepting runs:

$$\begin{aligned} \widehat{E}^t_{(s),pk}(2\hat{m}_1, 2\hat{r}_1, 0, 0) &= c'^2 c^{2e_1} \\ \widehat{E}^t_{(s),pk}(2\hat{m}_2, 2\hat{r}_2, 0, 0) &= c'^2 c^{2e_2} \end{aligned}$$

Multiplying the inverse of one of the encryptions on the other gives:

$$\begin{aligned} \widehat{E}^t_{(s),pk}(2(\hat{m}_1 - \hat{m}_2), 2(\hat{r}_1 - \hat{r}_2), 0, 0) &= c^{2(e_1 - e_2)} \\ &= \widehat{E}^t_{(s),pk}(2(e_1 - e_2)m, 2(e_1 - e_2)r, 0, 0) \end{aligned}$$

Since $2(e_1 - e_2)$ is prime to n by assumption, we can set

$$\bar{m} = (\hat{m}_1 - \hat{m}_2)(e_1 - e_2)^{-1} \bmod n^s = m$$

To get the value of r, we have the problem that we do not know the order τ of g, so we cannot find the inverse of $(e_1 - e_2)$. This leads to two scenarios, where only the first is possible under the Strong RSA assumption:

$(e_1 - e_2)|(\hat{r}_1 - \hat{r}_2)$**:** Here we can simply divide the numbers and we get that:

$$\bar{r} = (\hat{r}_1 - \hat{r}_2)/(e_1 - e_2) = r (\bmod \tau)$$

$(e_1 - e_2) \not| (\hat{r}_1 - \hat{r}_2)$**:** Here we find the value $x = \gcd(2(e_1 - e_2), 2(\hat{r}_1 - \hat{r}_2))$. The equation

$$\widehat{E}^t_{(s),pk}(2(\hat{m}_1 - \hat{m}_2), 2(\hat{r}_1 - \hat{r}_2), 0, 0) = \widehat{E}^t_{(s),pk}(2(e_1 - e_2)m, 2(e_1 - e_2)r, 0, 0)$$

implies that

$$g^{2(\hat{r}_1 - \hat{r}_2)} = g^{2(e_1 - e_2)r} = G^{2(e_1 - e_2)}$$

Now if we set $y_0 = 2(\hat{r}_1 - \hat{r}_2)/x$ and $y_1 = 2(e_1 - e_2)/x$ we have:

$$g^{y_0} = (-1)^b G^{y_1}$$

for some bit b that can be found by testing the 2 possibilities. Given the way y_0 and y_1 was defined, it is clear that they are relatively prime. This means we can find α, β such that $\gamma y_0 + \beta y_1 = 1$. Given these to values we can compute:

$$\begin{aligned} g = g^{\gamma y_0 + \beta y_1} &= (g^{y_0})^{\gamma}(g^{\beta})^{y_1} \\ &= ((-1)^b G^{y_1})^{\gamma}(g^{\beta})^{y_1} \\ &= (-1)^{b\gamma}(G^{\gamma})^{y_1}(g^{\beta})^{y_1} \\ &= (-1)^{b\gamma}(G^{\gamma}g^{\beta})^{y_1} \bmod n \end{aligned}$$

Since $(e_1 - e_2)$ does not divide $(\hat{r}_1 - \hat{r}_2)$, we have $x < 2(e_1 - e_2)$ and thus $y_1 > 1$. This means we have found a non-trivial root of $(-1)^{b\gamma}g$, which is a contradiction to the Strong RSA assumption.

This implies that we end up in the first case and thereby we find $\bar{m}$ and $\bar{r}$ that satisfy the equation

$$c = \widehat{E}^t_{(s),pk}(\bar{m}, \bar{r}, b_0, b_1)$$

for some choice of b_0 and b_1. □

Proof of Legal Encryption of Certain Plaintext

The protocol for legal encryptions can be altered to a protocol for proving that something is a legal encryption of a publicly known plaintext by setting $m' = 0$. This result in the following protocol:

Protocol for legal encryption of message m
Input: $n, g, h, c = (G, H), m \in \mathbb{Z}_{n^s}$.
Private input for P**:** $r \in \mathbb{Z}_\theta$, such that $c = \widehat{E}^t_{(s),pk}(m, r, b_0, b_1)$ for some b_0 and b_1.

1. P chooses a random r' in $\{0, ..., 2^{|\theta|+2k_2}\}$, where k_2 is a secondary security parameter (e.g. 160 bits). P sends $c' = (G', H') = \widehat{E}^t_{(s),pk}(0, r', 0, 0)$ to V.

2. V chooses e, a random k_2 bit number, and sends e to P.

3. P sends $\hat{r} = r' + er$ to V.

4. V checks that G, H, G', H' are prime to n, have Jacobi symbol 1 and that the equation $\widehat{E}^t_{(s),pk}(2em, 2\hat{r}, 0, 0) = (G'^2 G^{2e} \bmod n, H'^2 H^{2e} \bmod n^{s+1}) = c'^2 c^{2e}$ holds. V accepts if and only if this is the case.

This protocol is also sound and complete honest verifier zero-knowledge, which follows directly from the protocol above and the observation, that if c' is not the encryption of the plaintext 0, there is at most one e that can satisfy the last equation.

Lemma 2.9 *Protocol for* legal encryption of m *is complete, statistical honest verifier zero-knowledge, and under the Strong RSA assumption (conjecture 2.3) it satisfies that from any pair of accepting conversations (between* V *and any prover) of form* $(c', e_1, \hat{r}_1)$, $(c', e_2, \hat{r}_2)$ *with* $e_1 \neq e_2$*, one can efficiently compute* r*, such that* $c = \widehat{E}^t_{(s),pk}(m, r, b_0, b_1)$ *(for some* b_0, b_1*), provided* 2^{k_2} *is less than the smallest prime factor of* n*.*

Decryption Proof

To make the decryption share, the server calculates the value

$$d_i = G^{2\Delta\alpha_i} \bmod n$$

The server needs to prove that this was indeed what it submitted, but we have to allow a possible factor of -1, so we accept that $d_i = \pm G^{2\Delta\alpha_i}$, which is why the value d_i^2 is used in the Lagrange interpolation. What needs to be proven is that

$$\alpha_i = \log_g(h_i) = \log_{G^{4\Delta}}(d_i^2) \bmod p'q'$$

This can be done using a proof identical to that of Shoup's RSA threshold signatures [57] and therefore similar to the proof of correct decryption in section 2.4.1.

Proof: Given a hash function $\mathcal{H}$ that outputs a k_2 bit hash, pick a random $r \in \{0, ..., 2^{|n|+2k_2} - 1\}$ and calculate

$$\hat{g} = g^r \bmod n, \hat{G} = G^{4\Delta r} \bmod n,$$

$$c = \mathcal{H}(g, G^{4\Delta}, h_i, d_i^2, \hat{g}, \hat{G}), z = \alpha_i c + r$$

The proof is the pair (c, z).

Verification: For a proof to be accepted the following equation has to hold

$$c = \mathcal{H}(g, G^{4\Delta}, h_i, d_i^2, h_i^{-c} g^z \bmod n, d_i^{-2c} G^{4\Delta z} \bmod n)$$

This proof of correctness is sound and statistical zero-knowledge under the random oracle model. This is proven in Shoup's paper on practical threshold signatures [57] and is therefore omitted here.

Cross Public Key Proofs

Now we explore the possibility of proving relations on the plaintexts inside ciphertexts, that are either received or sent in the proof friendly cryptosystem. For the proofs to work, however, we require that the encryptions are valid in the proof friendly system. This means that all ciphertexts used in the proof have at some point been proven to be legal encryptions using one of the protocols above. Given an encryption:

$$c = \widehat{E}^t_{(s),pk}(m, r, b_0, b_1) = (G, H) = ((-1)^{b_0} g^r, E_{s,(n,n+1)}(m, (-1)^{b_1} h^r))$$

If we can find the m' and the r' such that

$$H = E_{s,(n,n+1)}(m', r')$$

then we can use the protocols from section 2.4.2 to prove various claims about the message or any linear combination of the messages by simply looking at the second value of the encryption.

For a sender it is easy to compute these values simply because he knows the m, b_1 and r used in the encryption and can calculate the two values as:

$$\begin{aligned} m' &= m \\ r' &= (-1)^{b_1} h^r \end{aligned}$$

For a receiver that receives c it is easy to compute m' simply by decrypting the message to get m. However, to compute the r' we need to use the trapdoor α to create the value:

$$\bar{r} = G^{4\Delta^2\alpha} = (g^r)^{4\Delta^2\alpha} = (g^\alpha)^{4\Delta^2 r} = h^{4\Delta^2 r} \bmod n$$

Now we can find r' by testing whether $r' = \bar{r}$ or $r' = -\bar{r}$ satisfies the equation:

$$(r')^{n^s}(n+1)^{m'} = H \bmod n^{s+1}$$

this will detemine the bit b_1 used in the encryption.

Now lets assume a player has two lists of ciphertexts: 1) a list of ciphertexts $c_1, ..., c_t$ sent by the prover under different public keys $pk_1, ..., pk_t$ (here $pk_i = h_i = g^{\alpha_i}$) and 2) a list $c_{t+1}, ..., c_{t+u}$ of received ciphertexts under the provers public key $pk = h^\alpha$. Then the above 2 computations can be used to find the values m'_i, r'_i for all $0 < i \leq t+u$ such that $H_i = E_{s,(n,n+1)}(m'_i, r'_i)$, where $(G_i, H_i) = c_i$. This also means that the prover can use the proofs from section 2.4.2 on any linear combination of the plaintexts in $c_1, ..., c_{t+u}$:

$$m = a_1 m'_1 + \cdots + a_{t+u} m'_{t+u}$$

by creating the two values

$$\begin{aligned} m' &= a_1 m'_1 + \cdots + a_{t+u} m'_{t+u} \\ r' &= (r'_1)^{a_1} \cdots (r'_{t+u})^{a_{t+u}} \end{aligned}$$

It can easily be verified that:

$$E_{s,(n,n+1)}(m', r') = H_1^{a_1} \cdots H_{t+u}^{a_{t+u}} \bmod n^{s+1}$$

an so the player can complete any of the protocols in section 2.4.2 using values on this form.

Using the multiplication proof from section 2.4.2 this can be extended to prove relations of any polynomial over $m_1, ..., m_{t+u}$. This is done by creating some intermediate encryptions with different multiplications of the messages.

2.7.3 Homomorphic Properties

The three cryptosystems $\overline{CS}$, $\overline{CS}^t$, and $\widehat{CS}^t$ defined above are all additive homomorphic, which means that two encryptions can be combined to create a new encryption of the sum of the plaintexts in the original encryptions. The proof friendly cryptosystem can be shown to be additive homomorphic by multiplying the values in ciphertexts pairwise (this also works for the other two cryptosystems):

$$\begin{aligned}\widehat{E}^t_{(s),pk}(m_0, r_0, b_{00}, b_{01})\widehat{E}^t_{(s),pk}(m_1, r_1, b_{10}, b_{11})&\\ &= (G_0, H_0)(G_1, H_1)\\ &= (G_0G_1, H_0H_1)\\ &= \widehat{E}^t_{(s),pk}(m_0 + m_1, r_0 + r_1, b_{00} \oplus b_{10}, b_{01} \oplus b_{11})\end{aligned}$$

It is also easy to multiply constants into the encryption by computing:

$$\widehat{E}^t_{(s),pk}(m, r, b_0, b_1)^c = (G_0^c, H_0^c) = \widehat{E}^t_{(s),pk}(cm, cr, cb_0, cb_1)$$

The additive homomorphic property also provides an easy way to re-randomize encryptions:

$$\widehat{E}^t_{(s),pk}(m, r, b_0, b_1)\widehat{E}^t_{(s),pk}(0, r', b'_0, b'_1) = \widehat{E}^t_{(s),pk}(m, r + r', b_0 \oplus b'_0, b_1 \oplus b'_1)$$

The homomorphism only works when the 2 encryptions use the same s. To get around this, one of the following transformations can be used either to increase or to decrease the s used for the encryption. However, both will change the message inside the encryption.

Given an encryption $(G, H) = \widehat{E}^t_{(s),pk}(m, r, b_0, b_1)$, we can transform it into an encryption using $s' > s$ by doing the following transformation:

$$\begin{aligned}(G', H') &= (G, H^{n^{s'-s}} \bmod n^{s'+1})\\ &= ((-1)^{b_0}g^r, ((-1)^{b_1}(h^r)^{n^s}(n + 1)^m)^{n^{s'-s}} \bmod n^{s'+1})\\ &= ((-1)^{b_0}g^r, (-1)^{b_1}(h^r)^{n^{s'}}(n + 1)^{mn^{s'-s}} \bmod n^{s'+1})\\ &= \widehat{E}^t_{(s'),pk}(mn^{s'-s}, r, b_0, b_1)\end{aligned}$$

If the encryption $(G, H) = \widehat{E}^t_{(s),pk}(m, r, b_0, b_1)$ needs to be transformed into an encryption using $s' < s$, we can do the following:

$$\begin{aligned}(G', H') &= (G^{n^{s-s'}} \bmod n, H \bmod n^{s'+1})\\ &= (((-1)^{b_0}g^r)^{n^{s-s'}} \bmod n, (-1)^{b_1}(h^r)^{n^s}(n + 1)^{m \bmod n^{s'}} \bmod n^{s'+1})\\ &= ((-1)^{b_0}g^{rn^{s-s'}} \bmod n, (-1)^{b_1}(h^{rn^{s-s'}})^{n^{s'}}(n + 1)^{m \bmod n^{s'}} \bmod n^{s'+1})\\ &= \widehat{E}^t_{(s'),pk}(m \bmod n^{s'}, rn^{s-s'}, b_0, b_1)\end{aligned}$$

Since the order of g is relatively prime to n the value $rn^{s-s'}$ will span just as big a subgroup of $\langle g\rangle$ as r alone.

Chapter 3

Anonymity Using Mix-nets

Privacy lost can never be regained
— David Chaum, Summer school in Cryptology '98, University of Aarhus.

In this chapter we look at a publicly verifiable length-flexible mix-net. This is achieved by using one of the cryptosystems from chapter 2 instead of El Gamal in one of the existing mix-nets. This results in a system that is both length-flexible and universally verifiable thereby improving previous results.

3.1 Introduction

3.1.1 Background

One possible application of homomorphic encryption is to build mix-nets. These are protocols used to provide anonymity for senders by collecting encrypted messages from several users and have a collection of servers process these, such that the plaintext messages are output in a randomly permuted order. A useful property for mix-nets is length-flexibility, which means that the mix-net is able to handle messages of arbitrary size. More precisely: all messages submitted to a single run of the mix-net must have the same length in order not to break the anonymity, this common length can be decided freely for each run of the mix-net without having to change any public-key information. This is especially useful for providing anonymity for e.g. e-mails. One way to achieve length-flexibility is to use hybrid mix-nets. These mix-nets use a public key construction to create keys for a symmetric cipher that is used for encrypting the bulk of the messages.

Two length-flexible hybrid mix-nets have been proposed. Ohkubo and Abe proposed a scheme in [4] in which verification of server behavior relies on a generic method by Desmedt and Kurosawa [34]. This results in a system, that is robust when at most the square root of the number of mix servers are corrupt. After this Juels and Jakobsson suggested in [46] that verification can be added by using message authentication codes (MACs), which are appended to the plaintext for each layer of encryption. This allows tolerating more corruptions at the expense of efficiency - for instance, the length of the ciphertext now depend on the number of mix servers as opposed to [4], and each server has to store

more secret material. Although the system is verifiable, it is not universally verifiable, which means that external observers cannot verify that everything was done correctly.

In [2] (with some minor pitfalls corrected in [3]), Abe introduced verifiable mix-nets based on a network of binary switching gates. This binary network was introduced Waksman [58] and can perform any permutation of the inputs. This mix-network is robust with up to half of the mix servers being controlled by an active and malicious adversary. One approach to make this length-flexible would be to exchange El Gamal with a verifiable length-flexible encryption scheme. The proof friendly cryptosystem in chapter 2 has the required properties.

3.1.2 Contribution

We combine the proof friendly cryptosystem from 2.7 with ideas from [2, 4] to construct a mix-net that has several desirable properties at the same time:

- ***Length-flexibility:*** The public key does not limit the size of plaintexts that can be encrypted and mixed efficiently. The length of ciphertexts in a specific mix have to be fixed or anonymity will be compromised.
- ***Length-invariance:*** The lengths of the keys and ciphertexts do not depend on the number of mix servers.
- ***Provable security:*** The system is provably secure in the random oracle model under the Decisional Composite Residuosity Assumption (conjecture 2.1) and composite DDH (conjecture 2.2).
- ***Universal verifiability:*** Anyone can verify the correctness of the output from the mix-net.
- ***Strong correctness:*** Messages submitted by malicious users cannot be changed once they have been submitted. This holds even in the case of helping malicious mix servers.
- ***Order flexibility:*** The mix servers do not need to be invoked in a certain order. This improves resilience to temporary server unavailability.

We note that all this is achieved by using public key encryption everywhere, which in the passive adversary case makes it less efficient than the Hybrid mix-nets that use symmetric key cryptography to encrypt the messages.

3.2 The Mix-net Model

The model used in this chapter relies on a secure bulletin board B. The bulletin board is assumed to function as follows: every player can write to B, and a message cannot be deleted once it is written. All players can access all messages written, and can identify which player each message comes from. This can all be implemented in a secure way, for instance using an already existing public key infrastructure and server replication to prevent denial of service attacks.

Besides the bulletin board there are some users $U_1, \cdots, U_u$, where u is the number of users. These users will post messages to B in an encrypted form. The purpose of the protocol is then to retrieve the plaintexts of these encryptions in such a way, that no one is able to see who posted it. This means that there should be no way to link the original encryptions with the plaintexts output by the protocol.

The last group of players in the mix-net model are the authorities (mix servers) $A_1, \cdots, A_w$, where w is the number of servers. The purpose of these authorities is to read a number of encryptions from B, perform a random permutation of these, and output the plaintexts of the encryptions. This is done in such a way, that unless all servers (or most in some schemes) cooperate, no one can link the input encryptions to the output plaintexts.

If we want to mix messages of different sizes (i.e. length-flexible) we have to ensure that for any single mix of messages, the messages posted on B have the same length. Otherwise one could match the sizes of the inputs and outputs to find some information on the permutation. For practical applications this means, that a fixed upper bound on the size of the plaintext space is decided for each mix, and all input messages for that mix have to be from the chosen message space. However, this bound can be chosen freely for each mix.

3.3 Adversaries

An adversary in [4] is defined by $(t_u, t_s)^{**}$, where the $*$ is either A for an active adversary or P for a passive adversary. The thresholds t_u and t_s are the maximal number of users and servers respectively, that can be controlled by the adversary. For example $(t_u, t_s)^{AP}$-adversary means, that the adversary can read and change any value for up to t_u users and view any value inside t_s servers. A passive adversary only observes the values passing a server or user, but does not try to induce values into the process or disrupt it. An active adversary can attack the protocol by changing any value or refuse to supply results in any part of the protocol. The adversary is assumed to be static, meaning that the users and servers being controlled by the adversary are decided in advance.

The mix-net in this chapter is safe against these adversaries of increasing strength (u is the number of users and w the number of servers):

- $(u-2, w-1)^{PP}$-adversary: Here the adversary can see any value passing through all but 1 server and all but 2 of the users.
- $(u-2, w-1)^{AP}$-adversary: The adversary can see any value passing through all but 1 server and see and change any value inside all but 2 users.
- $(u-2, \lfloor (w-1)/2 \rfloor)^{AA}$-adversary: This is the strongest adversary that can see and change any value passing through all but 2 users and less than half of the servers.

Compared to the length-flexible mix-net in [4], the first 2 adversaries are the same. However, in the case of an active adversary controlling the servers our scheme is improved from $(u-2, \mathcal{O}(\sqrt{w}))^{AA}$ to $(u-2, \lfloor (w-1)/2 \rfloor)^{AA}$.

3.4 Security of the Mix-net

We will use a strong version of correctness, so even if users are working together with servers, they will not be able to change the message once the mix has started.

Definition 3.1 (Strong Correctness) *Given x encrypted messages as input, where y of the encryptions are malformed, the mix-net will output a permutation of the $x - y$ messages with correct decryption, and discard all y malformed encryptions.*

Definition 3.2 (Anonymity) *Given a mix of x messages, and an $(t_u, t_s)^{**}$-adversary, the adversary should be unable to link any of the $x - t_u$ messages with any of the $x - t_u$ uncorrupted users, who sent them.*

Definition 3.3 (Universal Verifiability) *Given the public view of the protocol being all the information written to the bulletin board, there exists a poly-time algorithm V that accepts only if the output of the protocol is correct, and otherwise rejects.*

Definition 3.4 (Robustness) *Given an $(t_u, t_s)^{*A}$-adversary the mix-net protocol should always output a correct result.*

The mix-network presented can be shown to satisfy these definitions under the different adversaries.

Theorem 3.1 *The basic mix-network provides strong correctness and anonymity (and robustness) against an $(u - 2, w - 1)^{*P}$-adversary, where u is the number of users and w the number of servers.*

Theorem 3.2 *The mix-network with threshold decryption provides strong correctness, anonymity, universal verifiability and robustness against an $(u - 2, \lfloor (w - 1)/2 \rfloor)^{*A}$-adversary, where u is the number of users and w the number of servers.*

3.5 The System

The mix-network can be built using the threshold cryptosystem $\widehat{CS}^t$ from section 2.7. In the definition of the protocol a trusted third party (TTP) will be used, but this can be replaced with a distributed protocol by using the result from [5]. Since each mix uses a fixed s the specific encryption function $\widehat{E}^t_{(s),pk}(\cdot,\cdot,\cdot,\cdot)$ is used and not the general length-flexible $\widehat{E}^t_{pk}(\cdot,\cdot,\cdot,\cdot)$.

Key Generation: The TTP generates an RSA modulus $n = pq$ of length k bits, with $p = 2p' + 1$ and $q = 2q' + 1$, where p, q, p', q' are primes. It also selects a generator $g \in \mathrm{Q}_n$, the group of all squares of $\mathbb{Z}_n^*$. Depending on the model, the server picks the secrets the following way.

- **Passive adversary model:** For each mix server ($0 < i \leq w$) the TTP picks a random value $\alpha_i \in \mathbb{Z}_\tau$ and sets $\alpha = \sum_{0<i\leq w} \alpha_i \bmod \tau$. The public value is computed as $h = g^\alpha \bmod n$. The public key posted to B is $pk = (n, g, h)$ and the private key of A_i is α_i.
- **Active adversary model:** Here, the key generation takes place exactly as described for the threshold cryptosystem $\widehat{CS}^t$ defined in section 2.7. The public key $pk = (n, g, h)$ and the verification values $(h_1, ..., h_w)$ are posted to B. The private key α_i is given to A_i in a secure way.

Encryption: The s has to be fixed for each mix, so given a user U_i, that wants to submit a message $m_i \in \mathbb{Z}_{n^s}$, we do the following. Pick random values $r \in \mathbb{Z}_\theta, b_0, b_1 \in \{0, 1\}$. The ciphertext posted to B is

$$\widehat{E}^t_{(s),pk}(m, r, b_0, b_1) = ((-1)^{b_0} g^r \bmod n, \\ (-1)^{b_1}(h^{4\Delta^2 r} \bmod n)^{n^s}(n+1)^m \bmod n^{s+1})$$

The encryption needs to be non-malleable for the mix-net to satisfy the privacy requirements. To make this encryption non-malleable the user also submits a proof of knowledge of the values inside the encryption (i.e. the proof of correct encryption from section 2.7.2). Futhermore each mix needs to have a unique identifier that is included in the hash used to create the challenge for the non-interactive proof. This prevents an adversary from replaying the proof at some later mix.

Mixing phase: Before the mixing begins, any ciphertext (G, H) where either G or H has Jacobi symbol -1 will be discarded as being incorrect. Futhermore any ciphertext that has an invalid proof of knowledge or that appears twice in the input is discarded. While $I \neq \emptyset$ pick an $i \in I$ and let A_i make its permutation (mix) on the last correct output posted on B:

- **Passive adversary model:** Since the adversary is passive, A_i just does a random permutation and outputs a re-encryption for each of the ciphertexts (G, H) using random values b_0, b_1, r:

$$\begin{aligned}(G', H') &= (G(-1)^{b_0} g^r \bmod n, \\ &\qquad H(-1)^{b_1}(h^{4\Delta^2 r} \bmod n)^{n^s} \bmod n^{s+1}) \\ &= (G, H)\widehat{E}^t_{(s),pk}(0, r, b_0, b_1)\end{aligned}$$

- **Active adversary model:** Here verification is needed to satisfy the universal verifiability, correctness and robustness of the system. To do this, A_i picks a random permutation and creates a network of binary gates using the Waksman construction [58]. This network consists of $\mathcal{O}(u \log(u))$ binary gates and can create any permutation of the inputs. An example for the case $u = 8$ can be seen in figure 3.1. To make a proof that a complete permutation is done correctly, we just need to prove that each binary gate in the network has done

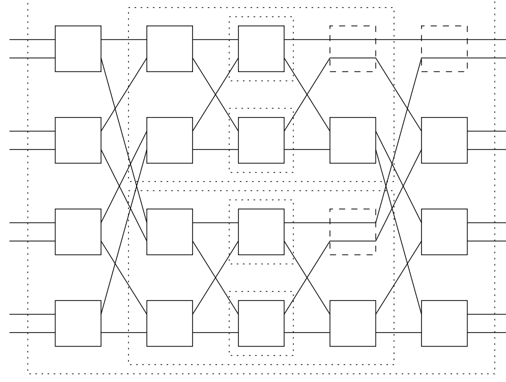


Figure 3.1: A network of binary gates that can be used to make any permutation of 8 inputs.

a permutation of the 2 inputs correctly. For each binary gate a bit B is defined (and $\bar{B} = 1 - B$), determining if the gate should pass the encryptions straight through the gate or switch them, depending on the permutation chosen for all the inputs. Each gate also has 2 ciphertexts (G_0, H_0) and (G_1, H_1) as input. The server chooses 6 random values: $x_0, x_1 \in \mathbb{Z}_N$ and $b_{00}, b_{01}, b_{10}, b_{11} \in \{0, 1\}$, and sets the 2 output ciphertexts for the gate to

$$\begin{aligned}
(G'_B, H'_B) &= (G_0(-1)^{b_{00}} g^{x_0}, H_0(-1)^{b_{10}} (h^{4\Delta^2 x_0})^{n^s}) \\
&= (G_0, H_0)\widehat{E}^t_{(s),pk}(0, x_0, b_{00}, b_{10}) \\
(G'_{\bar{B}}, H'_{\bar{B}}) &= (G_1(-1)^{b_{01}} g^{x_1}, H_1(-1)^{b_{11}} (h^{4\Delta^2 x_1})^{n^s}) \\
&= (G_1, H_1)\widehat{E}^t_{(s),pk}(0, x_1, b_{01}, b_{11})
\end{aligned}$$

To prove this is done correctly the server needs to prove that the B, satisfying the 2 equations above, really exist. This can be done by showing that the difference between (G'_B, H'_B) and (G_0, H_0) is a legal encryption of 0 (and likewise for $(G'_{\bar{B}}, H'_{\bar{B}})$ and (G_1, H_1)) for some $B \in \{0, 1\}$. The proof can be done using 4 concurrent runs of the legal encryption of the message 0 protocol from section 2.7.2. To tie the values together the technique from [20] is used, which simulates the one of the 2 statements it is unable to answer truthfully:

$$(G'_0 G_0^{-1}, H'_0 H_0^{-1}) \text{ and } (G'_1 G_1^{-1}, H'_1 H_1^{-1})$$

are legal encryptions of 0

or

$$(G'_1 G_0^{-1}, H'_1 H_0^{-1}) \text{ and } (G'_0 G_1^{-1}, H'_0 H_1^{-1})$$

are legal encryptions of 0

These proofs are posted to B along with the final outputs and intermediate encryptions. If the proof of the mix is incorrect or A_i refuses

to post a complete mix, any output from A_i is simply ignored, and the input used for A_i is used again for the next mix server.

When A_i has either posted a mix or refused to do so, it is removed: $I := I \backslash \{i\}$.

Decryption: After the mixing has been performed, the decryption of each of the output ciphertexts (G, H) needs to be performed. The removal of the h^r part is different depending on the model and is achieved in the following way:

- **Passive adversary model:** Each A_i makes a decryption share as $d_i = G^{\alpha_i} \bmod n$. The d_i values are posted to B and removed from the encryption in public:

$$H' = (H(\prod_{0<i\leq w} d_i)^{-n^s})^2 = (n+1)^{2m} \bmod n^{s+1}$$

- **Active adversary model:** Each A_i checks that at least $t+1$ servers have performed a legal mix, in which case at least 1 of them is honest, and it is safe to decrypt the encryptions. The value H' is computed in the same way as in $\widehat{CS}^t$ from section 2.7. Each A_i computes:

$$d_i = G^{2\Delta\alpha_i} \bmod n$$

and a proof that $\log_g(h_i) = \log_{G^{4\Delta}}(d_i^2)$, using the proofs from section 2.7.2. Each A_i posts d_i and the proof on B. A set S of the values d_i with a legal proof is chosen, and the d_i's are combined using Lagrange interpolation to create the exponent $4\Delta^2\alpha$:

$$d = \prod d_i^{2\lambda_i^S} = G^{4\Delta^2\alpha} = h^{4\Delta^2 r} \bmod n \quad \text{where } \lambda_i^S = \prod_{j\in S\backslash\{i\}} \Delta\frac{-j}{i-j}$$

and the $h^{4\Delta^2 r}$ can be removed:

$$H' = H^2 d^{-2n^s} = (n+1)^{2m} \bmod n^{s+1}$$

In both the passive and active model the value H' has the form $(n+1)^{2m}$ and the message is decrypted as $m = L_s(H' \bmod n^{s+1})/2 \bmod n^s$.

Remark 3.1 *In the above decription each server has been assigned one permutation network each. This is done for simplicity, while in practice only $t+1$ permutation networks are need. When using $t+1$ networks, any single server can only help with one of the $t+1$ permutation networks. This ensures that there is atleast one permutation network in which an adversary is not participating.*

The order of the mix servers can be chosen arbitrarily, which means that if A_i is unavailable when it is supposed to mix, A_j can do its mix. When A_i gets back again, it can perform its mix on the last output from one of the other mix servers as if nothing has happened.

The techniques from section 2.3.2 can be used to optimize all computations, except the last public exponentiation used to get H', from using $\mathcal{O}(s^3k^3)$ time to only $\mathcal{O}(s^2k^3)$.

3.6 Security Proofs

Before we prove the theorem lets introduce an alternative notion of semantic security:

Definition 3.5 *An adversary $\mathcal{A}$ against a public-key cryptosystem gets the public key pk, generated from security parameter k as input, and outputs two messages m_0, m_1. Then $\mathcal{A}$ is given an encryption under pk of either m_0 or m_1, and outputs a bit. Let $p_0(\mathcal{A}, k)$, respectively $p_1(\mathcal{A}, k)$, be the probability that $\mathcal{A}$ outputs 1 when given an encryption of m_0, respectively an encryption of m_1. Define the* advantage *of $\mathcal{A}$ to be $Adv(\mathcal{A}, k) = |p_0(\mathcal{A}, k) - p_1(\mathcal{A}, k)|$. The cryptosystem is* semantically secure *if for any probabilistic polynomial time adversary $\mathcal{A}$, $Adv(\mathcal{A}, k)$ is negligible in k.*

This definition is equivalent to definition 2.1, but for the proof below it is sufficient that it is implied by definition 2.1. This follows from a standard hybrid argument, where it is used that encryptions of m_0 and m_1 are both indistinguishable from encryptions of random messages.

Proof of theorem 3.1 and 3.2.

Robustness is a result of the setup of the threshold decryption. If any server refuses to do the mix, it is simply ignored, and the result can always be decrypted using the threshold decryption. The use of the bulletin board model with verification proofs at all steps of the protocol ensures universal verifiability. Strong correctness follows from the proof of correct encryption, which ensures that only valid encryptions are mixed, and from the binary gate proof, which forces the two input and the two output encryption to contain the same two messages. That only leaves anonymity:

Anonymity:
This is proven in two steps. First we show that if an adversary $\mathcal{B}$ can distinguish between $C = 0$ and $C = 1$ (here $\bar{C} = 1 - C$) such that:

$$y_C = \widehat{E}^t_{(s),pk}(0, r, b_0, b_1)$$

$$y_{\bar{C}} = \widehat{E}^t_{(s),pk}(x, r', b'_0, b'_1)$$

where x is a message of $\mathcal{B}$'s choice, then we can build an adversary $\mathcal{A}$ that can break the semantic security defined in definition 3.5, which implies an adversary against the semantic security from definition 2.1. Secondly we show that if an adversary $\mathcal{C}$ controlling the two input ciphertexts to a binary gate can guess the B used in the binary gate, then it is essentially solving the same problem as adversary $\mathcal{B}$.

Figure 3.2 is an adversary against the semantic security (as defined above) of the cryptosystem. We just need to verify its correctness. There are two possible values returned in step 4 of the algorithm in figure 3.2:

$c = \widehat{E}^t_{(s),pk}(0, r, b_0, b_1)$**:**

Adversary $\mathcal{A}$:

1. Get public key $pk = (n, g, h)$ and forward it to $\mathcal{B}$.
2. Get message x from $\mathcal{B}$.
3. Pass $0, x$ to the encryption oracle.
4. Get the encryption $c = \widehat{E}^t_{(s),pk}(m, r, b_0, b_1)$, where
   - $m = 0$ or
   - $m = x$.
5. Calculate $c' = \widehat{E}^t_{(s),pk}(x, r', b'_0, b'_1)c^{-1} \bmod n^{s+1}$ for random values r', b'_0, b'_1.
6. Send c, c' to $\mathcal{B}$.
7. Get the bit b from $\mathcal{A}$.
8. Return b.

Figure 3.2: Algorithm for an adversary $\mathcal{A}$ breaking the semantic security using the adversary $\mathcal{B}$ that can guess a permutation of two encryptions.

In this case we have:

$$\begin{aligned} c' &= \widehat{E}^t_{(s),pk}(x, r', b'_0, b'_1)c^{-1} \\ &= \widehat{E}^t_{(s),pk}(x, r', b'_0, b'_1)\widehat{E}^t_{(s),pk}(0, r, b_0, b_1)^{-1} \\ &= \widehat{E}^t_{(s),pk}(x, r', b'_0, b'_1)\widehat{E}^t_{(s),pk}(-0, -r, b_0, b_1) \\ &= \widehat{E}^t_{(s),pk}(x, r' - r, b'_0 \oplus b_0, b'_1 \oplus b_1) \end{aligned}$$

This is a random encryption of x and the encryptions we pass to $\mathcal{B}$ contains the plaintexts: $0, x$. So $\mathcal{B}$ is correct if $b = C = 0$ and so is $\mathcal{A}$.

$c = \widehat{E}^t_{(s),pk}(x, r, b_0, b_1)$**:**

In this case we have:

$$\begin{aligned} c' &= \widehat{E}^t_{(s),pk}(x, r', b'_0, b'_1)c^{-1} \\ &= \widehat{E}^t_{(s),pk}(x, r', b'_0, b'_1)\widehat{E}^t_{(s),pk}(x, r, b_0, b_1)^{-1} \\ &= \widehat{E}^t_{(s),pk}(x, r', b'_0, b'_1)\widehat{E}^t_{(s),pk}(-x, -r, b_0, b_1) \\ &= \widehat{E}^t_{(s),pk}(0, r' - r, b'_0 \oplus b_0, b'_1 \oplus b_1) \end{aligned}$$

This is a random encryption of 0 and the encryptions we pass to $\mathcal{B}$ contains the plaintexts: $x, 0$. So $\mathcal{B}$ is correct if $b = C = 1$ and so is $\mathcal{A}$.

Now we can look at an adversary against the binary gates. We will assume the adversary knows the values $m_0, m_1, r_0, r_1, b_{00}, b_{01}, b_{10}, b_{11}$ of the two input ciphertexts:

$$c_0 = (G_0, H_0) = \widehat{E}^t_{(s),pk}(m_0, r_0, b_{00}, b_{01})$$

$$c_1 = (G_1, H_1) = \widehat{E}^t_{(s),pk}(m_1, r_1, b_{10}, b_{11})$$

The binary gate outputs two ciphertexts:

$$c'_0 = (G'_0, H'_0) = \widehat{E}^t_{(s),pk}(m_B, r_2, b_{20}, b_{21})$$

$$c'_1 = (G'_1, H'_1) = \widehat{E}^t_{(s),pk}(m_{\bar{B}}, r_3, b_{30}, b_{31})$$

Given these two pairs of ciphertexts we can define the 4 differences as:

$$\begin{aligned} y_0 &= c'_0 c_0^{-1} & y_1 &= c'_1 c_0^{-1} \\ y_2 &= c'_1 c_1^{-1} & y_3 &= c'_0 c_1^{-1} \end{aligned} \tag{3.1}$$

If we have $B = 0$ we have (where $x = m_1 - m_0$):

$$\begin{aligned} y_0 &= \widehat{E}^t_{(s),pk}(0, r'_0, b'_{00}, b'_{01}) & y_1 &= \widehat{E}^t_{(s),pk}(x, r'_1, b'_{10}, b'_{11}) \\ y_2 &= \widehat{E}^t_{(s),pk}(0, r'_2, b'_{20}, b'_{21}) & y_3 &= \widehat{E}^t_{(s),pk}(-x, r'_3, b'_{30}, b'_{31}) \end{aligned}$$

and if $B = 1$ we have:

$$\begin{aligned} y_0 &= \widehat{E}^t_{(s),pk}(x, r'_0, b'_{00}, b'_{01}) & y_1 &= \widehat{E}^t_{(s),pk}(0, r'_1, b'_{10}, b'_{11}) \\ y_2 &= \widehat{E}^t_{(s),pk}(-x, r'_2, b'_{20}, b'_{21}) & y_3 &= \widehat{E}^t_{(s),pk}(0, r'_3, b'_{30}, b'_{31}) \end{aligned}$$

In the case where $B = 0$, the encryption y_0 is clearly a random encryption of 0 chosen by the mix server. Looking at this equation

$$y_1 = c'_1 c_0^{-1} = (y_2 c_1) c_0^{-1} = y_2 (c_1 c_0^{-1})$$

we can see that y_1 is the combination of a random encryption of 0 chosen by the mix server and the fixed encryption $(c_1 c_0^{-1})$ is chosen by the adversary. The encryption $(c_1 c_0^{-1})$ is an encryption of the value x, and since this is multiplied with a random encryption of 0, the value y_1 will be a random encryption of x that the adversary cannot control.

In the case where $B = 1$, we get that y_1 is a random encryption of 0. From equation 3.1 we get the following relation between y_0 and y_3:

$$y_0 = c'_0 c_0^{-1} = (y_3 c_1) c_0^{-1} = y_3 (c_1 c_0^{-1})$$

This means that y_0 is a random encryption of the message that was in the encryption $(c_1 c_0^{-1})$. So the two encryption y_0 and y_1 are random encryptions, with respect to c_0 and c_1, of 0 and x respectively. Here x is difference between the messages that the adversary controls, which means that he can choose any value of x. Given c_0, c_1, y_0 and y_1 the rest of the values in the binary gate are fixed:

$$\begin{aligned} c'_0 &= y_0 c_0 \\ c'_1 &= y_1 c_0 \\ y_2 &= c'_1 c_1^{-1} = y_1 c_0 c_1^{-1} \\ y_3 &= c'_0 c_1^{-1} = y_0 c_0 c_1^{-1} \end{aligned}$$

The difference between $\mathcal{B}$ and $\mathcal{C}$ is that $\mathcal{B}$ submits x, and $\mathcal{C}$ submits an encryption of x in the form of $(c_1 c_0^{-1})$. Both get back 2 encryptions y_0 and y_1 that are random encryptions with the restriction that one encrypts 0 and the other encrypts x. The goal for both of them is to guess which plaintext belong to which encryption. The only difference between the two adversaries is whether the encryption oracle get the message x or an encryption, but in both cases it responds with a random encryption of 0 and x, so this is irrelevant for the security.

If the adversary $\mathcal{C}$ does not control the input to the binary gate in the first place it will have strictly less information, and so it will have a smaller chance of succeeding.

This means that an adversary cannot gain any information on the permutation from the binary gates, and therefore his only chance is to break the encryption system. Decryption however can't be simulated, so in theory it might leak some information one the bits. For a way to change the protocol such that decryption can be simulated, see remark 3.2 below.

$\square$

Remark 3.2 *Decryption cannot be simulated as described in the protocol above, however there are two ways to fix this: 1) rerandomize the encryptions before decryption and 2) use the Cramer-Shoup variant [23] mentioned in section 2.5.1. The first option seems to require publicly verifiable secret sharing of some sort, so in most cases option 2 will be the most efficient choice of the two.*

Chapter 4

Secure On-line Voting

A citizen of America will cross the ocean to fight for democracy, but won't cross the street to vote in a national election.
— Bill Vaughan

This chapter concerns various on-line voting schemes. The general model uses a bulletin board which ensures that anyone can verify that everything is done correctly. During the chapter we will look at two flavors of elections, namely efficient large scale elections, which are useful for national elections with a large number of candidates and voters. The second flavor is board room elections. They are mostly useful for small groups of voters since messages have to be passed between each pair of voters. However, it does not require servers to decrypt the result, rather it becomes available as a result of the computations the voters perform. It is also possible to get perfect ballot secrecy in this setting, which means that any group of voters only learns what follows directly from their votes and the final result (which is clearly optimal in terms of privacy).

4.1 Introduction

4.1.1 Background

Voting schemes are one of the most important examples of advanced cryptographic protocols with immediate potential for practical applications. The most important goals for such protocols are

Privacy: only the final result is made public, no additional information about votes will leak.

Robustness: the result correctly reflects all submitted and well-formed ballots, even if some voters and/or possibly some of the entities running the election cheat.

Universal Verifiability: after the election, the result can be verified by anyone.

Other properties may be considered as well, such as receipt-freeness, i.e. voters are not able to prove the fact, that they voted for a particular candidate, thereby discouraging vote-buying or coercing.

Various fundamentally different approaches to voting are known in the literature: one may use blind signatures and anonymous channels [39], where the anonymous channels can be implemented using mix-nets (see [1,4] for instance) or based on some physical assumption. Another approach is to use several servers to count the votes and have voters verifiably secret share votes among the servers [21, 53]. Finally, one may use homomorphic encryption, where a voter simply publishes an encryption of his vote. Encryptions can be combined into an encryption of the result, and finally a number of decryption servers can cooperate to decrypt the result [22], assuming the private key needed for this is secret-shared among them.

Since anonymous channels are quite difficult to implement in practice, and verifiable secret sharing requires communication between a voter and all servers, the third method seems the most practical, and this chapter deals only with variants of this approach.

A large number of such schemes are known, but the most efficient one, at least in terms of the work needed from voters, is by Cramer, Gennaro and Schoenmakers [22]. This protocol in fact provides a general framework that allows usage of any probabilistic encryption scheme for encryption of votes, if the encryption scheme has a set of "nice" properties. In particular it must be homomorphic. The basic idea of this is straightforward: each voter broadcasts an encryption of his vote (by sending it to a bulletin board) together with a proof that the vote is valid. All the valid votes are then combined to produce an encryption of the result, using the homomorphic property of the encryption scheme. Finally, a set of trustees (who share the secret key of the scheme in a threshold fashion) can decrypt and publish the result.

Paillier pointed out already in [52], that since his encryption scheme is homomorphic, it may be applicable to electronic voting. In order to apply it in the framework of [22], however, some important building blocks are missing: one needs an efficient proof of validity of a vote, and also an efficient threshold variant of the scheme, so the result can be decrypted without allowing a single entity the possibility of learning how single voters voted.

In the following, we let L be the number of candidates, M the number of voters, w the number of decryption servers, and k the security parameter for the cryptosystem used. We assume for simplicity that each voter can vote for one candidate. In [22] a solution was given that may be based on any homomorphic threshold encryption scheme, if the scheme comes with certain associated efficient protocols. One example of this is El Gamal encryption. The ballot size in this scheme is $\mathcal{O}(\log M + b)$, where b is the block size (the size of the plaintext space) that the encryption scheme is set up to handle. The scheme was designed for the case of $L = 2$, and the generalization to general L given in [22] has complexity exponential in L for the decryption of the final result. Even for $L = 2$, a search over all possible election results is required to compute the final result. Therefore, this scheme does not scale well to large elections with many candidates.

In [10] and some of the solutions below use a variant of the approach from [22], but based on Paillier's cryptosystem. These are the first solutions that scale reasonably well to large elections, still the most efficient of these protocols produce ballots of size $\mathcal{O}(k \cdot \log(L))$. As long as $k > L \cdot \log(M)$, this is logarithmic in L, but for larger values of L and M it becomes linear in L, and each voter has to do $\Omega(\log(L))$ exponentiations using a modulus of length $L \cdot \log(M)$ bits. In a real application, one must assume, that voters typically have only rather limited computing power available (not state of the art), so the computation and communication needed for each voter is a rather critical parameter. On the other hand, decryption servers can be expected to be high-end machines connected by high-speed networks.

Thus for a large scale election, it is reasonable to consider the possibility of moving work away from voters at the expense of increased load on the servers. The central issue here is how much we can expect to reduce the size of ballots, since both communication and computational complexity for the voter is directly linked to this parameter. A moments reflection will show, that there is no fundamental reason why the size of a ballot should depend on M or be linear in L. Of course, a ballot must be at least $\log(L)$ bits long, since otherwise we cannot distinguish between the L candidates. Also, it would be unreasonable to expect the encryption to be secure, if the size of an encryption (a ballot) did not increase with the security parameter k. Thus a ballot size of $\mathcal{O}(k + \log(L))$ bits would be essentially optimal. In principle, this is easy to achieve: each voter V_i publishes an encryption of v_i (the id of the candidate he votes for), and the decryption servers use generic multi-party computation [41] to securely produce the result. This is always possible because the encryptions and the decryption key, which is shared among the servers, together determine the result and could be used to compute it efficiently, if they were public. Such a solution, however, would be much too inefficient to have any practical value. It would increase the complexity for the servers by a factor corresponding to at least the size of a Boolean circuit computing the decryption.

In [47] Kiayias and Yung introduced a new paradigm for electronic voting, namely protocols that are self-tallying, dispute-free and have perfect ballot secrecy (STDFPBS for short). This paradigm is suitable for e.g. boardroom elections where a (small) group of users want a maximally secure vote without help from external authorities. The main property is perfect ballot secrecy, which means that *any* coalition of voters (even a majority) can only get the information they can compute from the result of the election and their own votes, namely the tally of honest users votes. This is the best we can hope for, and it is the type of privacy that is actually achieved by paper based elections. Self-tallying means that as soon as all votes have been cast, no further interaction is needed to compute the result. It can be efficiently computed by just looking at all ballots, which can be done, even by a (casual) third party. Dispute-freeness means that no disputes between players can arise, because all faults are detected in public.

In [47] it is argued that STDFPBS elections cannot be achieved efficiently by traditional methods. For instance, large scale solutions are typically not of this type, because they assume that some set of authorities is available to

help with the election. The authorities typically share a secret key that can be reconstructed by a majority. In a small scale scenario we could let each voter play the role of an authority himself, but this would not give perfect ballot secrecy because a corrupt majority would know how *every* single voter voted. If we try to repair this by setting the threshold of the secret sharing scheme to be the total number of voters, then even a single fault will mean that the secret key is lost, and an expensive key generation phase would be needed.

In [47] STDFPBS elections are achieved for a yes/no vote by using constructs based on discrete log modulo a prime. This results in a tallying phase that needs to find a discrete log, which requires $\mathcal{O}(\sqrt{u})$ work when there are u voters. It also implies, that generalization to multi-way elections either results in larger ballots or much worse complexity for the tallying phase. Given earlier work on electronic voting, it is natural to speculate that this could be solved simply by using Paillier encryption instead. However, as noted in [47] this does not work, we would lose some essential properties of the scheme.

4.1.2 Related Work

In [37] voting was pointed out as a potential application for their cryptosystem. However, no suggestion was made for protocols to prove that an encrypted vote is correctly formed, something that is of course necessary for a secure election in practice.

In work done concurrently with and independent from ours, Stern, Baudron, Fouque, Pointcheval and Poupard [10] propose a voting scheme somewhat similar to ours. Their work can be seen as being complementary to ours in the sense, that their proposal is more oriented toward the system architectural aspects of a large scale election, and less toward optimization of the building blocks. To compare with their scheme, we first note that their modulus length k must be chosen such that $k > L \cdot \log(M)$. The scheme produces ballots of size $\mathcal{O}(k \cdot L)$. An estimate with explicit constants is given in [10], in which the dominating term in our notation is $9kL$.

Because our voting scheme uses the generalized Paillier cryptosystem, k can be chosen independently from L, M. In particular the scheme can scale to any size of election, even after the keys have been generated. However, if we choose k as in [10], i.e. $k > L \cdot \log(M)$, then the ballots we produce have size $\mathcal{O}(k \cdot \log(L))$. Working out the concrete constants involved, one finds that our complexity is dominated by the term $10k \log(L)$. So already for moderate size elections we have gained a significant factor in complexity compared to [10].

In [45] Hirt and Sako propose a general method for building receipt-free election schemes, i.e. protocols where vote-buying or -coercing is not possible, because voters cannot prove to others how they voted. Their method can be applied to make a receipt-free version of the scheme from [22]. It can also be applied to our scheme, with the same efficiency gain as in the non-receipt-free case.

When using the threshold version of our scheme, we assume for simplicity a trusted dealer for setting up the keys initially, and we assume that the modulus used is a safe prime product, similar to what is done in Shoup's paper [57].

In [32] Damgård and Koprowski propose techniques by which one can drop these restrictions from Shoup's scheme at the expense of an extra intractability assumption. The same idea can be easily applied to our scheme, thus producing a scheme without a trusted dealer and using a general RSA modulus. The threshold version of our scheme can also be used for *general* secure multi-party computation as shown by Cramer, Damgård and Nielsen in [19].

4.1.3 Contribution

The building blocks needed for the Paillier cryptosystem to be applicable to electronic voting were given in chapter 2. Thus we immediately get a voting protocol. In this protocol, the work needed from the voters is of the same order as in the original version of [22]. However, the work needed to produce the result is reduced dramatically, as we now explain. With the El Gamal encryption used in [22], the decryption process after a yes/no election produces $g^R \bmod p$, where p is prime, g is a generator and R is the desired result. Thus one needs to solve a discrete log problem in order to find the result. Since R is bounded by the number of voters M, this is feasible for moderate size M's. However, it requires $\Omega(\sqrt{M})$ multiplications and may certainly be something one wants to avoid for large scale elections. The problem becomes worse, if we consider an election where we choose between L candidates, $L \geq 2$. The method given for this in [22] is exponential in L in that it requires time $\Omega(\sqrt{M}^{L-1})$, and so is prohibitively expensive for elections with large L.

In the scheme we propose below, this work can be removed completely. Our decryption process produces the desired result directly. Moreover, we can easily scale to larger values of M and L without choosing new keys, just by going to a larger value of s.

We also give ways to efficiently implement constraints on voting that occur in real elections, such as allowing to vote for precisely l out of the L candidates, or to vote for up to l of them. In each of these schemes, the size of a single ballot is $\mathcal{O}(k \cdot L)$, where k is the bit length of the modulus used[1]. We propose a variant using a different technique, where ballots have size $\mathcal{O}(max(k, L\cdot\log(M))\cdot\log(L))$. Furthermore, this scheme requires only 1 decryption operation, even when $L > 2$.

In this chapter we also present a solution, that achieves ballot size $\mathcal{O}(k + \log(L))$ bits at the cost of each server having to broadcast $\mathcal{O}(M \cdot L \cdot (k + L \cdot \log(M)))$ bits. Most of this work can be done in a preprocessing phase, and only $\mathcal{O}(M \cdot (k + L \cdot \log(M)))$ bits need to be sent while the election is running. We assume the random oracle model, and that a static adversary corrupts less than $w/2$ servers and any number of voters. Then the protocol can be proved to be private, robust and verifiable, based on the semantic security of Paillier's public key system and the strong RSA assumption. We also present a variant with somewhat larger voter load, where the ballot size is $\log(L)\cdot(k+L)$ bits. This is still less than the previous Paillier-based solutions, as the communication per

[1] All complexities given here assume that the length of challenges for the zero-knowledge proofs is at most k.

server is $\mathcal{O}(M \cdot \log(\log(M))(k + L \cdot \log(M)))$ bits. Also here, preprocessing is possible leading to the same on-line cost as before. This variant can be proved secure in the random oracle model in the same sense as the previous variant, but assuming only semantic security of Paillier's public key system. Both variants can be executed in constant-round. None of the variants are receipt-free as they stand, but under an appropriate physical assumption, they can be made receipt-free using the techniques of [45].

Previous solutions based on the same assumption require each server to read each voters encrypted vote, process this, and broadcast a single piece of data. This results in communication that is linear in M, like in the systems we propose here. Thus, the extra cost for servers in our solution is that more rounds of interaction are required, and that the amount of communication is increased by a factor of L or $\log(\log(M))$.

The main new technique we use is to have voters work with a cryptosystem with block size $max(k, \log(L))$. The servers then securely transform this to encryptions in a related cryptosystem with block size $max(k, L \cdot \log(M))$, and compute the result using this second system. On top of this, we borrow some techniques from [19].

We note that optimal ballot size can also be achieved using the approach mentioned above based on anonymous channels, where the channels can be implemented using a mix-network. The mix-net hides the origin of a ballot. Therefore all ballots can be decrypted after mixing and vote counting becomes trivial. For some mix-net implementations we get communication complexity for the servers comparable to what we achieve here. Moreover, and perhaps more importantly, it seems to be inherent in the mix-net approach that mix servers do their work sequentially, i.e. each mix server can only act after the previous one has completed (part of) its work. By contrast, the threshold cryptography approach we use allows servers to complete the protocol in a constant number of rounds. Finally, using a mix-net, it is not clear that one can push most of the server work into a preprocessing phase, as we do here.

The final trade-off we present is of a completely different type. It relates more to practical security of elections. One of the worst potential weaknesses of electronic voting in practice is, that voters are likely to be non-expert computer users and most likely will use their own machines, home PCs, to cast votes, say over the Internet. Tools such as SSL plus signed applets, can be used to give reasonable assurance that the client software used for this is genuine. However, it is very difficult (some would say impossible) to make sure, that the user's machine is not infected by a virus. Such a virus could be used to monitor key strokes etc., and later transmit these to an adversary, who could then easily find out who the voter voted for. By contrast, it seems like a more reasonable assumption, that for instance a high-security server placed at some neutral site is not corrupted.

Motivated by this, we propose a solution with the following properties: privacy for the voter is ensured, even if his machine is completely monitored by an adversary, who can follow key strokes, screen image, mouse events, etc. Correctness of the result is ensured, assuming that a particular trusted party, who takes part in registering voters, behaves correctly (cheating will not allow

him to break the privacy, however). Whereas this party can in principle be held accountable and can be caught if he cheats, such verification is rather cumbersome. Hence, in practice, this solution trades trust in client machines against some amount of trust in a designated party. We note that a natural candidate for such a player often exists anyway in traditional manual voting schemes, and so in fact no "new" trust is needed - we discuss this in more detail later.

The basic idea of this solution is quite general. It can be combined with our first trade-off without significant loss of efficiency, but can also be applied to a very simple multi candidate election protocol, that can be based on Paillier encryption or on El Gamal, and requires the servers to do only L decryptions.

The last contribution in this chapter is to apply one of the cryptosystem from chapter 2 to construct STDFPBS elections. This results in a system, where the tallying phase reveals the result with a small number of additions, instead of $\mathcal{O}(\sqrt{u})$ multiplications as in [47]. This also shows that STDFPBS elections with all the essential properties can be based on Paillier encryption, thus solving a problem left open in [47]. Finally, it implies a natural and efficient generalization to multi-way elections.

4.2 Efficient Electronic Voting

4.2.1 Model and Notation

In [22], a general model for elections was used, which we briefly recall here: we have a set of voters $V_1, ..., V_M$, a bulletin board B, and a set of tallying authorities $A_1, ..., A_w$. The bulletin board is assumed to function as follows: every player can write to B, and a message cannot be deleted once it is written. All players can access all messages written, and can identify which player each message comes from. This can all be implemented in a secure way, for instance using an already existing public key infrastructure and server replication to prevent denial of service attacks. We assume that the purpose of the vote is to elect a winner among L candidates, and that each voter is allowed to vote for $l < L$ candidates.

In the following, $\mathcal{H}$ will denote a fixed hash function used to make non-interactive proofs according to the Fiat-Shamir heuristic. Also, we will assume throughout that an instance of the threshold cryptosystem CS_s^t, defined in section 2.4.1, is set up with a public key $pk = (n, s)$. The authorities A_i's are acting as decryption servers. We will assume that $n^s > M^L$, which can always be made true by choosing s or n large enough.

The notation $\text{Proof}_P(S)$, where S is some logical statement, will denote a bit string created by player P as follows: P selects the appropriate protocol from section 2.4.2 that can be used to interactively prove S. He computes the first message a in this protocol, computes $e = \mathcal{H}(a, S, ID(P))$ where $ID(P)$ is his user identity in the system and, taking the result of this as the challenge from the verifier, computes the answer z. Then $\text{Proof}_P(S) = (e, z)$. The inclusion of $ID(P)$ in the input to $\mathcal{H}$ is done in order to prevent vote duplication. To check such a proof, note that all the auxiliary protocols are such, that from S, z, c

one can easily compute what a should have been, had the proof been correct. For instance, for the protocol for n^s powers, the statement consists of a single number u modulo n^{s+1}, and the verifier checks that $z^{n^s} = au^e \bmod n^{s+1}$, so we have $a = z^{n^s} u^{-e} \bmod n^{s+1}$. Once a is computed, one checks that $e = \mathcal{H}(a, S, ID(P))$.

4.2.2 A Yes/No Election

A protocol for the case $L = 2$ is now simple to describe. This is equivalent to a yes/no vote and so each vote can be thought of as a number equal to 0 for no and 1 for yes:

1. Each voter V_i picks his vote $v_i \in \{0, 1\}$, he calculates $c_i = E^t_{s,pk}(v_i, r_i)$, where r_i is randomly chosen in $\mathbb{Z}^*_n$. He also creates

$$\text{Proof}_{V_i}(c_i \text{ or } c_i/(1+n) \text{ is an encryption of } 0)$$

   based on the 1-out-of-2 n^s'th power protocol from section 2.4.2. He writes the encrypted vote and proof to B.

2. Each A_j does the following:

   (a) Set $c = 1$.

   (b) For all voters V_i:

      - Check the proof written by V_i on B.
      - If it is valid set $c := cc_i \bmod n^{s+1}$.

   (c) Calculate the decryption share using c as input ciphertext according to the threshold decryption in section 2.4.1. The decryption share is written to B.

3. From the messages written by the A_j's, anyone can now reconstruct the plaintext corresponding to c (possibly after discarding invalid messages). Assuming for simplicity that all votes are valid, it is evident that

$$c = \prod_i E^t_{s,pk}(v_i, r_i) = E^t_{s,pk}(\sum_i v_i \bmod n^s, \prod_i r_i \bmod n^{s+1})$$

   So the decryption result is $\sum_i v_i \bmod n^s$, which is $\sum_i v_i$ since $n^s > M^L$.

Security of this protocol (in the random oracle model) can be proved based on the security results we have shown for the sub-protocols used, and based on semantic security of Paillier's encryption scheme. A more formal proof can be seen in [43], where Groth presents a full proof of security for the voting scheme according to the definition of Canetti [15].

4.2.3 A Multi-Candidate Election

There are several ways to generalize this to $L > 2$. Probably the simplest way is to hold L parallel yes/no votes as above. A voter votes 1 for the candidates he wants, and 0 for the others. This means that V_i will send L votes of the following form (where $j = 0, \cdots, L-1$):

$$c_{ij} = E^t_{s,pk}(v_{ij}, r_{ij}),$$

$$\text{Proof}_{V_i}(c_{ij} \text{ or } c_{ij}/(n+1) \text{ is an encryption of } 0)$$

To prove that he voted for exactly l candidates, he also writes to B the number $\prod_{j=0}^{L-1} r_{ij} \bmod n$. This allows the talliers to verify that $\prod_{j=0}^{L-1} E^t_{s,pk}(v_{ij}, r_{ij})$ is an encryption of l. This check is sufficient, since all individual votes are proved to be 0 or 1. It is immediate that decryption of the L results will immediately give the number of votes each candidate received.

The size of a vote in this protocol is seen to be $\mathcal{O}(Lsk)$, where k is the bit length of n, by simple inspection of the protocol. The protocol requires L decryption operations. As a numeric example, suppose we have $k = 1000, M = 64000, L = 64$, $s = 1$ and we use challenges of 80 bits in the proofs. Then a vote in the above system has size about 32 Kbyte.

We note that this easily generalizes to cases where voters are allowed to vote for *up to* l candidates: one simply introduces l "dummy candidates" in addition to the actual L. We then execute the protocol as before, but with $l + L$ candidates. Each voter places the votes he does not want to use on dummy candidates.

A more efficient method for large l is to add only 1 dummy candidate who is to receive all unused votes. Each voter must still prove that the product of all his encryptions decrypts to l. So given this proof, it is sufficient to prove that the number of votes on the dummy candidate is small enough. The number has to be small enough, so that a reduction modulo n^s cannot take place when the votes of this voter are added. This can be done by taking the bit string, representing the number of votes on the dummy candidate: $b_0...b_m$ where $2^m \leq l < 2^{m+1}$. The voter then makes encryptions $e_{ij} = E^t_{s,pk}(b_j 2^j, r_{ij})$ for all $0 \leq j \leq m$ and makes a proof for each of these:

$$\text{Proof}_{V_i}(e_{ij} \text{ or } e_{ij}/(1+n)^{2^j} \text{ is an encryption of } 0)$$

The votes for the dummy candidate can then be calculated as $c_{iL} = \prod_{j=0}^{m} e_{ij}$. Then it is verified as above that $\prod_{j=0}^{L} c_{ij}$ is the encryption of l. This only uses $L + 1$ blocks and $L + \log(l)$ proofs.

4.2.4 A variant with smaller vote size

If the parameters are such that $M^L < n^s$ and $l = 1$, then we can do significantly better than above. These conditions will be satisfied in many realistic situations, such as for instance in the numeric example above.

The basic idea is the following: a vote for candidate j, where $0 \leq j < L$, is defined to be an encryption of the number M^j. Each voter will create such

an encryption and prove its correctness as detailed below. When all these encryptions are multiplied we get an encryption of a number of the form $a = \sum_{j=0}^{L} a_j M^j \bmod n^s$, where a_j is the number of votes cast for candidate j. Since $M^L < n^s$, this relation also holds over the integers, so decrypting and writing a in M-ary notation will directly produce all the a_j's. It remains to describe how to produce an encryption hiding a number of form M^j, for some $0 \leq j < L$, and prove it is correctly formed. We do this in the following two subsections.

We note that this idea generalizes to $l > 1$, at some loss of efficiency. We simply allow each voter to cast l votes, each of the form just described. If we want to prevent voters from voting for the same candidate l times, we can use the homomorphic property to compute encryptions of all pairwise differences of votes, and the voter must prove that these are all non-zero. To show that m is non-zero, given the encryption $E^t_{s,pk}(m, r)$, the voter provides an encryption $E^t_{s,pk}(m^{-1} \bmod n^s, r')$ and uses the *multiplication-mod-n^s* protocol from section 2.4.2 to prove that the product of the two plaintexts is 1.

The case of $L = 2^{m+1}$

For simplicity, we will first describe how to prove correctness of a vote in the case where L is of the form $L = 2^{m+1}$ for some m, and treat the general case below. Let $b_0, ..., b_m$ be the bits in the binary representation of j, i.e. $j = b_0 2^0 + b_1 2^1 + \cdots + b_m 2^m$. Then clearly we have $M^j = (M^{2^0})^{b_0} \cdot \ldots \cdot (M^{2^m})^{b_m}$. Each factor in this product is either 1 or a power of M. This is used in the following algorithm for producing the desired proof (where P denotes the prover):

1. P computes encryptions $c_0, ..., c_m$ of $(M^{2^0})^{b_0}, ..., (M^{2^m})^{b_m}$. For each $i = 0, ..., m$ he also computes
$$\text{Proof}_P(c_i/(1+n) \text{ or } c_i/(1+n)^{M^{2^i}} \text{ is an encryption of } 0)$$
2. Let $F_i = (M^{2^0})^{b_0} \cdot \ldots \cdot (M^{2^i})^{b_i}$, for $i = 0, ..., m$. P computes an encryption f_i of F_i, for $i = 1, ..., m$. We set $f_0 = c_0$. Now, for $i = 1, ..., m$, P computes
$$\text{Proof}_P(\text{Plaintexts corr. to } f_{i-1}, c_i, f_i \text{ satisfy}$$
$$F_{i-1} \cdot (M^{2^i})^{b_i} = F_i \bmod n^s \text{ }),$$
based on the *multiplication-mod-n^s protocol*. The encryption f_m is the desired encryption of the vote M^j.

It is straightforward to verify from c_i, f_i and all the proofs computed, that f_m is an encryption of a number of form M^j. Furthermore, simply because there are $m+1$ encryptions $c_0, .., c_m$ each determining one bit of j, it is clear that $0 \leq j < 2^{m+1} = L$.

It is straightforward to see that a vote in this system will have length $\mathcal{O}(sk \cdot \log(L))$ bits (still assuming, of course, that $M^L < n^s$).

With parameter values as in the numeric example before, a vote will have size about 7 Kbyte, a factor of almost 5 better than the previous system. Moreover, we need only 1 decryption operation as opposed to L before.

The case of general L

If L is not of the nice form we assumed above, we may attempt to adapt the above solution as follows: first define m by: 2^{m+1} is the smallest 2-power with $2^{m+1} > L$, and then run the above protocol with no further changes. There are two drawbacks to this idea: first, it allows voters to vote for non-existing candidates, namely j's for which $L \leq j < 2^{m+1}$, and second this also implies that we must have $M^{2^{m+1}} < n^s$, otherwise we may get overflow when votes are added, and the result will be incorrect. If we could prevent voters from voting for non-existing candidates, we would only need $M^L < n^s$, so this simple-minded solution may force us to have a block length larger than what is strictly necessary, in the worst case almost twice as large.

One way to get around this is to add an extra step to the verification of a vote where, given the encryptions $c_0, ..., c_m$ determining the bits of j, the voter proves in zero-knowledge that $j < L$.

To this end, first recall that we defined $j = b_0 2^0 + b_1 2^1 + \cdots + b_m 2^m$, and that for each encryption c_i that is provided, it is shown that it encrypts $M^{b_i 2^i}$. Define $\beta_i = (M^{2^i} - 1)^{-1} \bmod n^s$. It is now easy to see that

$$\begin{aligned} c_i' &= ((c_i(1+n)^{-1})^{\beta_i} \bmod n^{s+1}) \bmod n^2 \\ &= (c_i(1+n)^{-1})^{\beta_i \bmod n} \bmod n^2 \end{aligned}$$

is an encryption of b_i in CS_1^t. Furthermore a verifier can compute this value without interaction from already public information. Going to CS_1^t means, that the complexity of the protocol to follow becomes independent of s. From this point there are several ways to proceed. We sketch one simple option here:

Let L be represented by bits $B_0, ..., B_m$. We can now exploit the following fact:

$$j < L \text{ iff } \exists\, i, \text{ such that} B_m = b_m, \cdots, B_{i+1} = b_{i+1}, B_i = 1, b_i = 0$$

Notice that $d_i = ((2B_i - 1)(2b_i - 1) + 1)/2$ is a binary value that is 1, if $B_i = b_i$ and 0 otherwise. Since B_i is public, the verifier can compute an encryption of d_i from c_i' without interaction. Clearly, the product $D_i = d_m \cdots d_{i+1} B_i(1 - b_i)$ is 1 precisely if i is an index confirming that $L > j$. It is also easy to see that by providing encryptions of the values $(d_m d_{m-1})$, $(d_m d_{m-1} d_{m-2}), ..., (d_m \cdots d_1)$ and of the D_i's, the prover can show that the encryptions of the D_i's contain correct values, using $2m$ multiplication proofs. Finally, the prover needs to show that one of the $D_i = 1$ for some i. This can be done by a trivial generalization of the one-of-two protocol, we showed earlier, to a one-of-$(m-1)$ protocol. In total, this solution will have size $\mathcal{O}(k \cdot \log(L))$ bits (assuming that $M^L < n^s$). This is asymptotically the same as before, but with a larger constant. We note that in [6] Lipmaa et al. have recently proposed a conceptually simpler solution for general L, which is more efficient than ours by a constant factor.

4.3 Client/Server Trade-Offs

4.3.1 The Minimal Vote Election Scheme

In this section we introduce a scheme in which ballots are of essentially minimal size. This requires that a transformation of the votes is performed by the tally servers to a larger representation of the vote. From the transformed vote the result of the election can be found using the homomorphic properties as seen in the previous section and in [22].

Needed Properties

In the reduction of the voter load we need a pair of public key cryptosystems $\mathcal{CS}_1$ and $\mathcal{CS}_2$ with their respective encryption and decryption functions $\mathcal{E}_1$, $\mathcal{E}_2$, $\mathcal{D}_1$, and $\mathcal{D}_2$. An encryption of m in $\mathcal{CS}_i$ under public key pk using random input r will be denoted $\mathcal{E}_i(m, r)$. The public key is left out, because it is kept fixed at all times once generated. We will also often suppress r from the notation for simplicity. N_1 and N_2 will denote the size of the plaintext space for $\mathcal{CS}_1$ and $\mathcal{CS}_2$. The two cryptosystems should satisfy:

- **Semantically secure:** Both $\mathcal{CS}_1$ and $\mathcal{CS}_2$ are semantically secure.
- **$\mathcal{CS}_2$ is a threshold system:** The private key in $\mathcal{CS}_2$ can be shared among w decryption servers, such that any minority of servers have no information on the key, whereas any majority of servers can cooperate to decrypt a ciphertext while revealing no information other than the plaintext.
- **$\mathcal{CS}_2$ is homomorphic:** There exists an efficiently computable operation that can be applied to two ciphertexts to yield an encryption of the sum of the two plaintexts, that is, we have:

  $$\mathcal{E}_2(m_1 \bmod N_2)\mathcal{E}_2(m_2 \bmod N_2) = \mathcal{E}_2(m_1 + m_2 \bmod N_2)$$

  Furthermore, given $\alpha \in \mathbb{Z}_{N_2}, \mathcal{E}_2(m)$ it is easy to compute an encryption $\mathcal{E}_2(\alpha m \bmod N_2)$.
- **$\mathcal{CS}_2$ supports MPC multiplication:** There exists an interactive protocol, denoted MPC (Multi-Party Computation) multiplication, that the decryption servers can execute on two encryptions. The protocol produces securely a random encryption containing the product of the corresponding plaintexts, in other words, we can produce $\mathcal{E}_2(m_1 m_2 \bmod N_2, r_3)$ from $\mathcal{E}_2(m_1 \bmod N_2, r_1)$ and $\mathcal{E}_2(m_2 \bmod N_2, r_2)$ without revealing information about m_1 or m_2.
- **Interval Proofs:** There exists a zero-knowledge proof of knowledge (that can be made non-interactive in the random oracle model), such that having produced $\mathcal{E}_i(m)$, a player can prove in zero-knowledge that m is in some given interval I. For optimal efficiency we will need that the length of the proof corresponds to a constant number of encryptions. For the

special case of $I = 0, ..., N_i$ $(i = 1, 2)$, this just amounts to proving that you know the plaintext corresponding to a given ciphertext.

- **Transformable:** There exists a number $B \leq N_1$, s.t. given an encryption $\mathcal{E}_1(m, r)$, where it is guaranteed that $m \leq B$, there is an interactive protocol for the decryption severs producing as output $\mathcal{E}_2(m, r)$, without revealing any extra information.

- **Random Value Generation:** The decryption servers can cooperate to generate an encryption $\mathcal{E}_2(R)$, where R is a random value unknown to all servers.

- **Vote size:** $L \leq B \leq N_1$ so that votes for different candidates can be distinguished and encryptions be transformed.

- **Election size:** Let $j = \lceil \log_2(M) \rceil$. We need that $(2^j)^L < N_2$ to ensure that we do not get an overflow, when the final result is computed.

- **Factorization of N_2:** All prime factors of N_2 are super-polynomially large in the security parameter.

We do not want to give the impression this set-up is more general than it really is. We know only one efficient example of a system with the above properties, this example is described below. However, we stick to the above abstract description to shield the reader from unnecessary details, and to emphasize the essential properties we use.

Example 1

If we pick $\mathcal{CS}_1 = CS_s^t$ and $\mathcal{CS}_2 = CS_{s'}^t$ such that $s \leq s'$, the above properties are satisfied. This means that $N_1 = n^s$ and $N_2 = n^{s'}$.

Theorem 4.1 *Given $\mathcal{CS}_1 = CS_s^t$ and $\mathcal{CS}_2 = CS_{s'}^t$, the above properties are satisfied under the DCRA assumption (conjecture 2.1).*

Proof.

- **Semantically secure:** under the DCRA both $\mathcal{CS}_1$ and $\mathcal{CS}_2$ are semantically secure.

- $\mathcal{CS}_2$ **Homomorphic:** $CS_{s'}^t$ is homomorphic and we have that

$$\mathcal{E}(m)^\alpha \bmod n^{s'+1} = \mathcal{E}(\alpha m \bmod n^{s'})$$

- $\mathcal{CS}_2$ **supports MPC multiplication:** an efficient protocol is shown in [19], requiring each server to broadcast a constant number of encryptions.

- **Interval proofs:** the proof construction in section 4.3.4 constructs the required proof using communication equivalent to a constant number of encryptions.

- **Random value generation:** the decryption servers do the following: each server $0 < i \leq w$ chooses a random $R_i \in \mathbb{Z}_{N_2}$. The values $\mathcal{E}_2(R_i)$ are published, followed by zero-knowledge proofs that R_i is known by server i. These proofs can be done using the multi-party Σ-protocol technique from section 6 of [19] allowing the zero-knowledge proofs to be done concurrently in a non malleable way.

  We then form $\mathcal{E}_2(R) = \mathcal{E}_2(\sum_i R_i) = \mathcal{E}_2(R_1) \cdots \mathcal{E}_2(R_w)$. Thus R is random and unknown to all servers.

- **Transformable:** an encryption in $\mathcal{CS}_1$ can be transformed to an encryption in $\mathcal{CS}_2$ by using the method described below. This method requires that the bound B on the input message satisfies $\log(B) \leq \log(N_1) - k_2 - \log(w) - 2$, where k_2 is a secondary security parameter ($k_2 = 128$ for instance).

- **Vote size:** we need $L \leq B$, which as mentioned above means $\log(L) \leq \log(N_1) - k_2 - \log(w) - 2$. For most realistic values of k, k_2, L, w this will be satisfied even with $s = 1$, but otherwise s can always be increased.

- **Election size:** $M^L < N_2 = n^{s'}$ can always be satisfied by choosing s' large enough.

- **Factorization of N_2:** we have $N_2 = n^{s'} = (pq)^{s'}$, and p, q must of course be large to have any security at all.

□

We now show how to transform the ciphertext $\mathcal{E}_1(m)$ from $\mathcal{CS}_1$ to $\mathcal{CS}_2$, where it is known that $0 \leq m \leq B$. Our transformation will work if $\log(B) \leq \log(N_1) - k_2 - \log(w) - 2$.

The crucial observation is that a ciphertext $\mathcal{E}_1(m)$ in $\mathcal{CS}_1$ can always be regarded as a ciphertext in $\mathcal{CS}_2$, simply by thinking of it as a number modulo $n^{s'+1}$. It is not hard to see that as a $\mathcal{CS}_2$ encryption, it is an encryption of a number $m' \in \mathbb{Z}_{n^{s'}}$ with $m' = m \bmod n^s$. This is not good enough since we want $m = m' \bmod n^{s'}$. All we know is that $m' = m + tn^s \bmod n^{s'}$ for some t we cannot directly compute. To get around this, we mask m with some random bits so that we can find t by decryption, as detailed below.

The masking can be done in 2 ways:

- **Trusted Third Party:** A trusted third party generates a random value R of size $log(B) + k_2$. The trusted third party reveals the value $\mathcal{E}_2(R)$.

- **MPC approach:** The servers each generate a value R_i of length $log(B) + k_2$ bits, reveal $\mathcal{E}_2(R_i)$ and prove they have done so using an interval proof. This should be done using the multi-party Σ-protocol technique of [19]. All encryptions with correct proofs are combined using the homomorphic property to get

$$\mathcal{E}_2(R) = \Pi_{i \in I} \mathcal{E}_2(R_i) = \mathcal{E}_2(\Sigma_{i \in I} R_i)$$

where I is the set of servers that supplied a correct proof. This means that R is at most $w2^{\log(B)+k_2}$.

Note that the condition on B and R ensures that $m + R < N_1$.

1. We consider the encryption $c = \mathcal{E}_1(m)$ as a ciphertext c in $\mathcal{CS}_2$. As noted above, this will be the encryption $\mathcal{E}_2(m + tn^s \bmod n^{s'}, r)$ for unknown t and r.

2. Now let $c' = c \cdot \mathcal{E}_2(R)$.

3. The servers decrypt c' to get a message $m + R + tn^s \bmod n^{s'}$. Since we have $m + R < N_1$, we can find $m + R$ and t just by integer division. And if at least one server has chosen its R_i at random, information on m will be statistically hidden from the servers, since R is at least k_2 bits longer than m.

4. We now set $c'' = c\mathcal{E}_2(-tn^s, 1)$. Due to the homomorphic properties this is equal to $\mathcal{E}_2(m)$.

Preparation

The preparation phase requires the generation of the 2 cryptosystems $\mathcal{CS}_1$, $\mathcal{CS}_2$ with key distribution for threshold decryption in $\mathcal{CS}_2$. We also need a publicly known polynomial of degree $L - 1$ which satisfies the equation:

$$f(i) = M^i \bmod N_2 \qquad \forall i : 0 \leq i < L$$

By assumption N_2 has only very large prime factors. Hence any difference of form $i - j$ where $0 \leq i, j < L$ is invertible modulo N_2, and this is sufficient to ensure that f can be constructed using standard Lagrange interpolation.

The next and last part of the preparation has to be done once for each election. For each voter, the servers generate a random encryption $\mathcal{E}_2(R)$ as described earlier. Then we execute $L-2$ MPC multiplications to get encryptions $\mathcal{E}_2(R^j)$ for $j = 1, ..., L - 1$.

Voting

The voter generates a vote for candidate i by making an encryption $\mathcal{E}_1(i)$ and an interval proof, that it is the encryption of a value in the interval $\{0, ..., L-1\}$.

Transformation

When the servers receive the vote as a ciphertext in $\mathcal{CS}_1$, they have to transform it into a corresponding vote in $\mathcal{CS}_2$, that can be added together to give a meaningful result. This is done by transforming $\mathcal{E}_1(i)$ to $\mathcal{E}_2(M^i)$. This has to be done for each vote and can be done in the following way:

1. We transform $\mathcal{E}_1(i)$ into $\mathcal{E}_2(i)$.

2. The servers decrypt $c = \mathcal{E}_2(i)\mathcal{E}_2(R)$ to get $z = i + R$. It follows that $i^j = (z - R)^j$, and this can be rewritten using the standard binomial expansion. The result is that $i^j = \alpha_0 + \alpha_1 R + ... + \alpha_j R^j$ for publicly known values $\alpha_0, ..., \alpha_j$. Hence encryptions $\mathcal{E}_2(i^j)$ can be computed without interaction from the encryptions $\mathcal{E}_2(R^j)$ from the preparation phase, using the homomorphic property. From these encryptions, we can, using the polynomial f computed in the preparation, construct an encryption $\mathcal{E}_2(f(i))$, still with no further interaction. The result of this satisfies $\mathcal{E}_2(f(i) \bmod N_2, r) = \mathcal{E}_2(M^i \bmod N_2, r)$.

Calculating the Result

Now we can combine all the transformed votes using the homomorphic property of $\mathcal{CS}_2$ and decrypt the result. This will give a value of the form:

$$\sum v_i M^i \qquad \forall i : 0 \leq v_i < M$$

Since M is the number of voters, an overflow of $v_i \bmod M$ cannot have occurred, and since $M^L < N_2$, the number of votes on the i'th candidate will be v_i.

Complexity

From the voters point of view the computational (modular multiplications) and communicational complexity (bits) of this protocol will be $\mathcal{O}(\log(L) + k)$. This is within a constant of the smallest possible.

The decryption servers work depends on the cryptosystems used, and can only really be compared in the number of usages of the primitives: transformations (from $\mathcal{CS}_1$ to $\mathcal{CS}_2$), decryptions, MPC multiplications, and random value generations.

In the preparation, we generate M random values and do $M(L-2)$ MPC multiplications. During the election we do M transformations from $\mathcal{CS}_1$ to $\mathcal{CS}_2$ and M decryptions. An additional decryption is needed to get the result. To calculate the powers of R in the preprocessing, $\mathcal{O}(L)$ rounds of communication are needed. Constant round solutions can also be devised using techniques from [8], but the total communication will be larger. The protocol for the election itself is constant round.

4.3.2 An Alternative System

Here we look at an alternative scheme that requires more work for the voter, but the work required by the tallying servers can be reduced compared to the previous scheme for some parameter values.

Needed Properties

In this trade off scheme we also need a pair of cryptosystems $\mathcal{CS}_1$ and $\mathcal{CS}_2$ with properties as described earlier, except for two changes:

- **Zero-knowledge proofs**: Interval proofs are not needed for this scheme. Instead we need that a player can generate an encryption $\mathcal{E}_1(v)$ and prove in zero-knowledge that $v \in \{2^0, 2^1, ..., 2^{L-1}\}$. For the example of Paillier based encryption, a protocol for this purpose is given in section 4.2.4.
- **Vote Size**: In this scheme, we need $2^L \leq B \leq N_1$ instead of $L \leq B$.

Preparation

The preparation phase requires the generation of the 2 cryptosystems $\mathcal{CS}_1$, $\mathcal{CS}_2$ with key distribution for threshold decryption in $\mathcal{CS}_2$.

In preparation of each election, a pair of values has to be generated for each voter (recall that we defined j to be minimal, such that $2^j > M$):

- An encryption of some random R: $\mathcal{E}_2(R \bmod N_2)$.
- The inverse of R raised to the j'th power: $\mathcal{E}_2(R^{-j} \bmod N_2)$.

These values are generated before the election so that the result of the election is more efficiently computed, when the votes start to arrive. The values can be generated with one of these 2 methods:

- **Trusted third party**: The trusted third party generates the 2 encryptions.
- **MPC approach**: The servers cooperate on generating a random encryption $\mathcal{E}_2(R)$. Using the inversion method from [8] the value $\mathcal{E}_2(R^{-1})$ is generated[2]. Then the servers use the MPC multiplication $\mathcal{O}(\log(j))$ times to get $\mathcal{E}_2(R^{-j})$.

Voting

The voter generates a vote for candidate i by setting $v = 2^i$, making $\mathcal{E}_1(v)$ and a proof, that it is the encryption of a message from the set $\{2^0, ..., 2^{L-1}\}$.

Transformation

The goal of the transformation is to compute $\mathcal{E}_2((v)^j)$ from $\mathcal{E}_1(v)$ and can be done as follows:

1. The encryption of the vote v is transformed to $c = \mathcal{E}_2(v)$ in $\mathcal{CS}_2$.
2. The servers perform an MPC multiplication of $c = \mathcal{E}_2(v)$ and $\mathcal{E}_2(R)$ to get $c' = \mathcal{E}_2(vR \bmod N_2)$.
3. The servers decrypt c' to get $vR \bmod N_2$, which reveals no information of v since R is chosen at random (note that by assumption on N_2, both v and R are prime to N_2 except with negligible probability).

[2]This is done by generating another encryption of a random value R' in the same way as the first. Then compute the MPC multiplication of the 2 and decrypt it to get $RR' \bmod N_2$. This is inverted and encrypted again. Then this is MPC multiplied with $\mathcal{E}_2(R')$ again to get $\mathcal{E}_2((RR')^{-1}R' \bmod N_2) = \mathcal{E}_2(R^{-1} \bmod N_2)$

4. The servers raise vR to the j'th power in public and make an encryption of this, $c'' = \mathcal{E}_2((vR)^j \bmod N_2, 1)$ (we use a default value of 1 for the random input to encryption, no randomness is needed here).

5. The servers make an MPC multiplication of c'' and $\mathcal{E}_2(R^{-j} \bmod N_2)$ to get the transformed encryption of the vote

$$\mathcal{E}_2(v^j \bmod N_2, r) = \mathcal{E}_2((2^j)^i \bmod N_2, r)$$

Calculating the Result

To calculate the result the transformed votes are combined using the homomorphic property of $\mathcal{CS}_2$, and the resulting ciphertext is decrypted. The plaintext from the decryption will have the form:

$$\sum v_i (2^j)^i \qquad \forall i : 0 \leq v_i < 2^j$$

Since $2^j > M$, where M is the number of voters, an overflow cannot have occurred for a single candidate, and the whole election cannot have caused a overflow since $(2^j)^L < N_2$. The number of votes on the i'th candidate is v_i.

Complexity

The communication needed from the voter is now $\mathcal{O}(L+k)$, plus the size of the proof of correctness for the encryption (which in the Damgård-Jurik scheme will have size $\mathcal{O}(\log(L))$ encryptions using the techniques from [26]).

If a trusted third party is used, then there is no precomputation for the tally servers, but otherwise they have to generate the pair of values. To generate the inverses we need 1 random value generation, 2 MPC multiplications and 1 decryption, and for calculating the j'th power we need at most $2\log_2(j)$ multiplications, which means that we use a total of $M(\log_2(\log_2(M))+2)$ MPC multiplications, M decryptions and M random values.

For the election itself, the number of transformations we need from $\mathcal{CS}_1$ to $\mathcal{CS}_2$ is M. In the protocol we use a decryption when raising each vote to the j'th power, so we need $M+1$ decryptions. And finally we need a total of $2M$ MPC multiplications.

The preparation can be done in $\mathcal{O}(\log(\log(M)))$ rounds, while the protocol after preparation is constant round.

In comparison with the first scheme, we see that the voters do more work here, and the complexity of the election after preparation is comparable, but slightly lower in the first scheme. The main difference is that the complexity of the preparation is $\mathcal{O}(ML)$ MPC multiplications in the first scheme, and $\mathcal{O}(M \cdot \log(\log(M)))$ in the second. Another difference is that the first scheme requires ML encryptions to be stored between preparation and election, while the second scheme requires only $2M$ encryptions.

4.3.3 Protecting Clients Against Hackers

How can a voter be protected against a curious person who has full access to his computer during an election? In this section we look at a way to trade trust in the security of the client computer against trust in a third party. We first describe the basic idea on a high level and then give two ways to implement the idea.

We assume that we have a trusted third party (TTP) (we discuss later in which sense he has to be trusted). The TTP will for each voter choose a random permutation π permuting the set $0, 1, ..., L-1$. He then privately (and possibly by non-electronic means) sends a list containing for each candidate i, the candidate's name and $\pi(i)$. When using his own (or any) client machine to cast his vote, the voter decides on a candidate - say candidate number i, finds his name on the list, and tells the client software that he votes for candidate $\pi(i)$. The client software could simply present a list of numbers from 0 to $L-1$ to choose from, without any corresponding names. The client software sends an encryption of $\pi(i)$ to the tally servers.

At the same time as π is generated, the TTP also sends to the tally servers an encryption of π. Using this encryption, the servers can transform the encryption of $\pi(i)$ into an encryption of i, and the election result can then be computed using the homomorphic properties as usual.

As for security of this, consider first correctness: as we have described the system, we clearly have to trust that the TTP encrypts the correct permutation for each voter to the servers. If not, the result will be incorrect. Note, however, that the TTP cannot decrypt the encryption of $\pi(i)$ sent from the client machine, so it cannot manipulate the permutation and be certain to favor a particular candidate. If the TTP was suspected of foul play against a particular voter, the information held by the voter could be verified against the encryption of π, and then cheating would always be caught. However, since this is a rather cumbersome procedure, it shouldn't happen very often, and so some amount of trust has to be invested in the TTP.

As for privacy, clearly an attacker monitoring the client machine gets no information on who the voter voted for, by the random choice of π. Furthermore, even if the TTP pools its information with a minority of the severs, they cannot break the privacy of the voter, unless they break the encryption. A breach of privacy would require both that the TTP is corrupt, and that it participates in an attack where client machines are infected.

In practice, who might play the role of the TTP? As an example, in many countries, there is an authority which, prior to elections and referendums, sends by private paper mail a card to every eligible voter, and this card must be used when casting a vote. Such an authority could naturally play the role of the TTP and simply print the information about π on the card sent out. In other countries voters must contact a government office to get registered. In this case the permutation could be generated on the fly, and the information handed directly to the voter.

For the first implementation of this idea, we use a cryptosystem $\mathcal{CS}$ with encryption and decryption functions $\mathcal{E}, \mathcal{D}$ and plaintext space of size N.

Needed Properties

Here we need less assumptions because we do not need to transform the votes between different cryptosystems.

- **Semantically secure:** $\mathcal{CS}$ is semantically secure.
- $\mathcal{CS}$ **is a threshold system:** as defined earlier.
- $\mathcal{CS}$ **is homomorphic:** as defined earlier.
- $\mathcal{CS}$ **supports MPC multiplication:** as defined earlier.
- **Zero-Knowledge proofs:** we need that a player can generate an encryption $\mathcal{E}(v)$ and prove in zero-knowledge that $v \in \{M^0, ..., M^{L-1}\}$. For the example of Paillier based encryption, a protocol for this purpose is given in section 4.2.4.
- **Election size**: To ensure that the final result is correct we need $M^L < N$.
- **Factorization of** N**:** We assume that N has only very large prime factors, so that factoring N is infeasible.

Preparation

The TTP picks a random permutation π for each voter and gives the $\pi(i)$ values to the voter as described above. Then the TTP generates a polynomial of degree $L-1$ for each of the voters with the property that

$$f(M^i) = M^{\pi^{-1}(i)} \bmod N \quad \forall i : 0 \leq i < L$$

If doing this by Lagrange interpolation fails, this can only be because some number less than N was found to be non-invertible modulo N, which implies N can be factored. Since this was assumed infeasible, the construction fails with negligible probability. The L coefficients of the polynomial are then encrypted to produce encryptions $c_0, ..., c_{L-1}$, and these are given to the tallying servers.

The tally servers for each voter generates a random encryption $\mathcal{E}(R)$ and compute encryptions of the powers $\mathcal{E}(R^2), ..., \mathcal{E}(R^{L-1})$.

Voting

To vote for candidate i the voter gives $\pi(i)$ to the client machine, which makes an encryption $\mathcal{E}(M^{\pi(i)})$ and appends a zero-knowledge proof, that one of $M^0, ..., M^{L-1}$ was encrypted.

Transformation

We need to transform the vote $\mathcal{E}(M^{\pi(i)})$. First we use the encryptions of powers of R from the preparation to compute encryptions $\mathcal{E}(M^{2\pi(i)}), ..., \mathcal{E}(M^{(L-1)\pi(i)})$. This is done the same way as in the minimal vote scheme in section 4.3.1 and requires only one decryption and local computation. From this and $c_0, ..., c_{L-1}$,

the servers can clearly use the homomorphic property and $\mathcal{O}(L)$ MPC multiplication to make an encryption $\mathcal{E}(f(M^{\pi(i)}))$. If the TTP participates, it can be done much more efficiently. Since the TTP knows the coefficients of f, it can produce $\mathcal{E}(f(M^{\pi(i)}))$ from $\mathcal{E}(M^{2\pi(i)}),...,\mathcal{E}(M^{(L-1)\pi(i)})$ by only local computation and prove in zero-knowledge to the servers, that this was correctly done. The proof is straightforward to construct using techniques from [19][3].

We then have

$$\mathcal{E}(f(M^{\pi(i)})) = \mathcal{E}(M^{\pi^{-1}(\pi(i))}) = \mathcal{E}(M^i)$$

which is what we wanted.

Combination

The result can then be found using the homomorphic addition of the transformed votes to get a number of the form:

$$\sum v_i M^i \qquad \forall i : 0 \leq v_i < M$$

Since M is the number of voters and $M^L < N$ an overflow cannot have occurred, and the number of votes on the i'th candidate will be v_i.

Complexity

Since we do not have any reduction in the size of the cryptosystem the voters communication and computational complexities are $\mathcal{O}(L \log M + k)$ plus the size of the proof that the vote has the right form.

For the tallying servers, the complexity of both preparation and election is comparable to the minimal scheme in case the TTP participates in the election. Otherwise we will need $\mathcal{O}(ML)$ MPC multiplications during the election itself.

Combination with Minimal Votes

Since the scheme we just presented is similar to the minimal vote scheme, it is straightforward to combine the two. This only adds the cost of transforming the vote from $\mathcal{CS}_1$ to $\mathcal{CS}_2$. The polynomial must now have the form

$$f(i) = M^{\pi^{-1}(i)} \bmod N$$

and the voter sends an encryption of form $\mathcal{E}_1(\pi(i))$ as his encrypted vote.

An Alternative Implementation

A very simple way to implement multi candidate elections from homomorphic encryption is as follows: the voter produces encryptions $c_1,...,c_L$, where $c_i = \mathcal{E}(1)$, if he votes for candidate i and all other encryptions contain 0. He proves

[3] In [19], a zero-knowledge protocol was given by which a player can prove that a committed constant was correctly "multiplied into" a given encryption, and this is exactly what we need here.

in zero-knowledge that each c_j encrypts 0 or 1, and opens $c = c_1 \cdots c_L$ to reveal 1, in order to prove he voted for one candidate. The tally servers can then combine all 0/1 votes for each candidate separately using the homomorphic property and decrypt. This method places a quite large workload on voters, but on the other hand it can be based on El Gamal encryption as well as on Paillier, and it is the only known way in which elections with large L can be efficiently based on El Gamal encryption (the method from [22] is exponential in L).

It is straightforward to apply the client protection method to this voting scheme. The trusted third party TTP generates and communicates a permutation to each voter as described above. Then to encrypt a permutation π for the tally servers, the TTP will generate an $L \times L$ permutation matrix M_π, representing π^{-1} in the standard way, and publish encryptions of each entry in M_π. We let $\mathcal{E}(M_\pi)$ denote this (ordered) set of encryptions. The voter will now send a set of encryptions $c_1, ..., c_L$, where $c_{\pi(i)} = \mathcal{E}(1)$. Since the TTP knows the entries of M_π, he can, using only local computations and the homomorphic property, produce random encryptions $c'_1, ..., c'_L$, such that $c'_i = \mathcal{E}(1)$ and all the others contain 0. This is done by applying M_π to the vector of encryptions and multiplying by random encryptions of 0. Since $\mathcal{E}(M_\pi)$ was made public, he can then prove in zero-knowledge to the servers, that this was correctly done using techniques from [19]. Finally the computation of the final result can be completed as above.

4.3.4 Interval Proofs for Paillier Encryptions

Given a Paillier encryption $\mathcal{E}(m, r)$ (computed modulo n^{s+1}), we sketch here an efficient method to prove in zero-knowledge that m is in some given interval I. The protocol in section 4.2.4 provides an interval proof, but it needs to supply an encryption of every bit in the binary expansion of m. We want a more efficient method, where only a constant number of encryptions need to be send. In the following, *opening an encryption* $\mathcal{E}(m, r)$ means revealing m, r.

In [9] Boudot gives an efficient method for proving that a committed number lies in a given interval. The protocol requires sending only a constant number of commitments and is zero-knowledge in the random oracle model that we also use here. It assumes that the number has been committed to, using a commitment scheme with some specific properties. The scheme proposed by Fujisaki and Okamoto [40] will suffice, assuming the strong RSA assumption (see conjecture 2.3). See [9] for a short description of the commitment scheme and associated protocols. It should be noted, that there are some technical problems with the proof of soundness for the associated protocols given in [40], but these problems have recently been fixed in [24]. The modulus n used for Paillier encryption can also serve as part of the public key for the commitment scheme in Boudot's protocol. In addition, we need two elements $g, h \in \mathbb{Z}_n^*$ of large order, such that g is in the group generated by h. The prover must not know the discrete logarithm of g base h or vice versa. We assume that g, h are generated as part of the procedure that sets up n and shares the private key among the decryption servers. A commitment to m in this scheme using

random input r is $Com(m,r) = g^m h^r \bmod n$.

Now, the basic idea is the following: given $\mathcal{E}(m, r_1)$, the prover provides a commitment $Com(m, r_2)$, proves that the commitment contains the same number as the encryption, and then uses Boudot's protocol to prove that $m \in I$. The only missing link here is how to show, that the same number m is contained in encryption and commitment. This can be done using the following protocol:

Protocol for equality of message
Input: n, g, h, c_1, c_2.
Private input for P: $m \in I, r_1 \in \mathbb{Z}_n^*, r_2 \in \mathbb{Z}_\theta$, such that $c_1 = \mathcal{E}(m, r_1)$ and $c_2 = Com(m, r_2)$.

1. Let T be the maximal bit-length of m (based on the interval I). P chooses at random u, an integer of length $T + 2k_2$, where k_2 is a secondary security parameter. P also chooses $r_1' \in \mathbb{Z}_n^*$ and r_2' a $|\theta| + 2k_2$ bit number. He sends $a_1 = \mathcal{E}(u, r_1')$ and $a_2 = Com(u, r_2')$ to V.
2. V chooses e, a random k_2 bit number, and sends e to P.
3. P opens the encryption $a_1 \cdot c_1^e \bmod n^{s+1}$ and the commitment $a_2 \cdot c_2^e \bmod n$ by revealing $z_m = u + em$, $z_1 = r_1^e r_1' \bmod n$ and $z_2 = er_2 + r_2'$.
4. V checks that $a_1 \cdot c_1^e = \mathcal{E}(z_m, z_1) \bmod n^{s+1}$ and $a_2 \cdot c_2^e = Com(z_m, z_2) \bmod n$. V accepts if and only if this is the case.

This protocol can be made non-interactive in the standard way using a hash function and the Fiat-Shamir heuristic. Then it is also statistically zero-knowledge in the random oracle model.

What we have done is combine two already known protocols for proving knowledge of the contents of an encryption and a commitment, respectively. When we prove soundness of this protocol using a standard rewinding argument, the fact that we use the same challenge e and the same response z_m in both cases will ensure, that the prover must know one single value, which is inside both the encryption and the commitment.

Lemma 4.1 *Protocol for* equality of message *is complete, statistical honest verifier zero-knowledge, and under the Strong RSA assumption (conjecture 2.3) it satisfies that from any pair of accepting conversations (between V and any prover) of form $(a_1, a_2, e, z_m, z_1, z_2)$, $(a_1, a_2, e', z_m', z_1', z_2')$ with $e \neq e'$, one can efficiently compute m, r_1, r_2, such that $c_1 = \mathcal{E}(m, r_1)$ and $c_2 = Com(m, r_2)$, provided 2^{k_2} is less than the smallest prime factor of n.*

Proof. For completeness we can verify the equations tested by the verifier:

$$a_1 \cdot c_1^e = \mathcal{E}(u, r_1')\mathcal{E}(m, r_1)^e = \mathcal{E}(u + em, r_1' r_1^e) = \mathcal{E}(z_m, z_1) \bmod n^2$$

and

$$\begin{aligned} a_2 \cdot c_2^e &= Com(u, r_2') \cdot Com(m, r_2)^e = g^u h^{r_2'} (g^m h^{r_2})^e = g^{u+em} h^{r_2' + er_2} \\ &= Com(u + em, r_2' + er_2) = Com(z_m, z_2) \bmod n \end{aligned}$$

The values em and er_2 are statistically hidden by the values u and r_2', respectively. We can make a statistically close simulation by picking $z_m \in \{0, ..., 2^{T+2k_2} - 1\}$, $z_1 \in \mathbb{Z}_n^*$, and $z_2 \in \{0, ..., 2^{|\theta|+2k_2} - 1\}$ at random. Then we can make a_1, a_2 as

$$a_1 = \mathcal{E}(z_m, z_1)c_1^{-e}$$
$$a_2 = Com(z_m, z_2)c_2^{-e}$$

This can easily be verified to be statically close to the usual distribution.

For the claim that we can recover m, r_1, and r_2 we can look at:

$$\mathcal{E}(z_m, z_1) = a_1 \cdot c_1^e$$
$$\mathcal{E}(z_m', z_1') = a_1 \cdot c_1^{e'}$$

dividing these out and reducing modulo n we get:

$$\mathcal{E}(z_m - z_m', z_1/z_1') = c_1^{e-e'} = \mathcal{E}((e - e')m, r_1^{e-e'}) \bmod n^{s+1} \tag{4.1}$$

This means that:

$$(z_1/z_1')^{n^s} = (r_1^{e-e'})^{n^s} \bmod n$$

Since $e - e'$ is prime to n by the assumption on 2^{k_2}, choose α, β such that $\alpha n^s + \beta(e - e') = 1$. Let $\bar{c_1} = c_1 = r_1^{n^s} \bmod n$ and set $\bar{r}_1 = \bar{c_1}^{\alpha}(z_1/z_1')^{\beta} \bmod n$. We get that

$$(\bar{r}_1)^{n^s} = (\bar{c_1}^{\alpha}(z_1/z_1')^{\beta})^{n^s} = (r_1^{\alpha n^s}(r_1)^{\beta(e-e')})^{n^s} = (r_1)^{n^s} = c_1 \bmod n$$

So we have found the random value used in the encryption.

Using formula 4.1 we can find m as $\hat{m} = (z_m - z_m')(e - e')^{-1} \bmod n^s$, which follows from $z_m - z_m' = (e - e')m$. In fact, $e - e'$ will have to divide $z_m - z_m'$ or we can get a non-trivial root in exactly the same way as it is done for r_2 below.

To find r_2 we need to use the Strong RSA assumption. We have the following two equations:

$$Com(z_m, z_2) = a_2 \cdot c_2^e$$
$$Com(z_m', z_2') = a_2 \cdot c_2^{e'}$$

We can remove the message from the commitments by computing

$$\begin{aligned} h^{z_2 - z_2'} &= Com(0, z_2 - z_2') = Com(z_m - z_m', z_2 - z_2')g^{z_m' - z_m} \\ &= c_2^{e-e'} g^{z_m' - z_m} = (h^{r_2})^{e-e'} = h^{(e-e')r_2} \end{aligned}$$

The proof can now be split into two cases, one where $e - e'$ divides $z_2 - z_2'$ and one where it does not. Only the first case can happen, or we get a contradiction to the Strong RSA assumption. The two cases goes as follows:

$(e - e')|(z_2 - z_2')$: Here we can simply divide the numbers to get:

$$\bar{r}_2 = (z_2 - z_2')/(e - e') = r_2$$

$(e-e') \not| (z_2 - z_2')$: Here we find the value $x = \gcd(e-e', z_2 - z_2')$. If we set $y_0 = (z_2 - z_2')/x$ and $y_1 = (e-e')/x$ we have that:

$$h^{y_0} = h^{y_1 r_2} = Com(0, r_2)^{y_1} = \hat{c}_2^{y_1}$$

Given the way y_0 and y_1 were defined, it is clear that they are relatively prime. This means we can find α, β such that $\alpha y_0 + \beta y_1 = 1$. Given these two values we can compute:

$$\begin{aligned} h = h^{\alpha y_0 + \beta y_1} &= (h^{y_0})^\alpha (h^\beta)^{y_1} = (\hat{c}_2^{y_1})^\alpha (h^\beta)^{y_1} = (\hat{c}_2^\alpha)^{y_1} (h^\beta)^{y_1} \\ &= (\hat{c}_2^\alpha h^\beta)^{y_1} \bmod n \end{aligned}$$

Since $(e-e')$ does not divide $(z_2 - z_2')$, we have that $x < (e-e')$ and thus that $y_1 > 1$. This means that we have found a non-trivial root of h, which is a contradiction to the Strong RSA assumption.

□

4.4 Self-Tallying Elections with Perfect Ballot Secrecy

In this section, we will show how to make more efficient self-tallying elections with perfect ballot secrecy based on the cryptosystem $\widehat{CS}^t$ introduced in section 2.7. The values b_0 and b_1, that are used in the encryption, have been omitted in this section for simplicity, and the encryption is changed slightly because we don't need the threshold decryption. The power of h don't need the factor $4\Delta^2$, which leads to this encryption:

$$\widehat{E}^t_{(s),(n,g,h)}(m,r) = (G, H) = (\pm g^r \bmod n, \pm h^r (n+1)^m \bmod n^{s+1})$$

The decryption function is simply

$$m = \mathrm{dLog}_s(G^{-2\alpha} H^2)/2 \bmod n^s$$

Here we briefly sketch the main properties of the scheme, but for a more in-depth explanation the reader is referred to [47]:

Correctness: the result output is the correct result with respect to the correctly formed input ballots.

Privacy: the vote contained in each ballot is kept secret. This is usually achieved by trusting in some threshold setup of servers or by voter to voter communication.

Fairness: no part of the tally is revealed to anyone before the election is completed. This requires some trust in the bulletin board.

Universal Verifiability: anyone can verify the result of the election given access to the bulletin board. This even holds for casual third parties who have not participated in the protocol.

Self-Tallying: there are no decryption servers that do a threshold decryption at the end of the protocol. Instead the result becomes known as a result of the computations the voters made on the bulletin board.

Dispute-Freeness: dispute-freeness simply means that no parties can get into a fight about who is dishonest. This follows from the bulletin board model and the universal verifiability, since everyone will see every message posted and can verify the values themselves.

Perfect Ballot Secrecy: perfect ballot secrecy is the strongest type of privacy available in an election. It means, that any group of voters will only learn what is computable from their own votes and the final result. To find out who an honest voter voted for, requires all the remaining voters to team up (or at least enough voters so that the remaining votes are all for the same candidate).

Corrective Fault-Tolerance: in the case of a fault, it is possible to correct the current computation to get around the fault. This means, that if a player is caught trying to cheat, it is possible to remove him from the computation and finish the computation.

The system uses the bulletin board model described in section 3.2, and it is assumed that a safe prime product n and a generator $g \in \mathrm{Q}_n$ is set up in advance, so that the cryptosystem $\widehat{CS}^t$ from section 2.7 can be used.

The modulus n can be generated once and for all. One option is to let a trusted third party do this. Note, that since the factorization of n is never needed, not even in shared form, a trusted party solution can be quite acceptable. This could be achieved using a secure hardware box, which is destroyed after n has been generated.

Another option is to use a distributed protocol such as [5] or a generic multi-party computation. Note that protocols for this purpose can be set up such that no proper subset of the players can find the factors of n. In other words, we still ensure perfect ballot secrecy even if the players generate n themselves. This comes at the expense of possibly having to restart the key generation if faults occur, but this cost cannot be avoided if we need to handle dishonest majorities and is consistent with the way corrective fault tolerance is defined in [47].

The element g can be generated by jointly generating some random value $x \in \mathbb{Z}_n^*$ (with Jacobi symbol -1 according to remark in section 2.7.2) and then defining g as $g = x^2 \bmod n$, which will be in Q_n.

The bulletin board also participates in the protocol to ensure that none of the actual voters will know the result before they vote. With a self-tallying scheme, this type of fairness cannot be achieved without such trust (see [47]). One may think of the bulletin board as a party that must vote 0 (so it will not influence the result), and is trusted to submit its vote only after all players have voted. The bulletin board, however, does not have to participate in every step of the protocol. It will only participate in: 1) the registration phase, where it registers its public key, 2) the error correction of the ballot casting, where it has

some encrypted values it needs to reveal (this step can actually be skipped if the protocol is modified as described in the remark in section 4.4.3), and 3) the post ballot casting step, where it reveals its 0 vote, thereby enabling everyone to calculate the result.

4.4.1 Setup Phase

The setup phase consists of two tasks. First the voter registration and then the initialization of the voting system itself. In the registration phase voters that want to participate in the election register on the bulletin board. After all voters are registered, the voters need to set up the values to be used in the protocol. Since voters can be malicious in this part of the protocol, there is an error correction step to correct any problems encountered.

Voter Registration

Voter i chooses the private key α_i at random in $\mathbb{Z}_\theta$, where θ is the value defined in section 2.3.1. Then voter i computes the value $h_i = g^{\alpha_i} \bmod n$. The voter registers by posting the public key $pk_i = (g, h_i)$ on the bulletin board. Let R be the set of all registered voters and for simplicity let's assume that $R = \{1, 2, ..., u\}$. To ensure fairness, the bulletin board also generates a public key pk_0 and posts it on the bulletin board (we set $R_0 = R \cup \{0\}$).

Initialization

Each voter $i \in R$ picks random values $s_{ij} \in \mathbb{Z}_{n^s}$ for each $j \in R$ and random $r_{ij} \in \mathbb{Z}_\theta$ for each $j \in R_0$. The value s_{i0} is set to $-\sum_{j\in R} s_{ij} \bmod n^s$, which ensures that $\sum_{j\in R_0} s_{ij} = 0 \bmod n^s$.

The voter i publishes the encryptions

$$c_{ij} = (G_{ij}, H_{ij}) = \widehat{E}^t_{(s),pk_j}(s_{ij}, r_{ij})$$

for all $j \in R_0$ along with a proof, that these are indeed legal encryptions and the sum of the plaintexts is 0 modulo n^s.

To prove these are legal encryptions, the proof from section 2.7.2 is used. To prove that the sum of the plaintexts in the encryptions $(G_{i0}, H_{i0}), ..., (G_{iu}, H_{iu})$ is 0, it is enough to look at the product $H_{i0} \cdots H_{0u}$. The resulting value is

$$H_{i0} \cdots H_{iu} = (h_0^{r_{i0}} \cdots h_u^{r_{iu}})^{n^s} (n+1)^{s_{i0}+\cdots+s_{iu}}$$

which is an n^s'th power iff $\sum_{j\in R_0} s_{ij} = 0 \bmod n^s$. The protocol for n^s'th powers from section 2.4.2 can be used to prove this, since the voter knows an n^s'th root of this number, namely $h_0^{r_{i0}} \cdots h_u^{r_{iu}}$.

Error Correction of the Initialization

Let B_1 be the set of voters, that either do not supply all the encryptions or supply invalid proofs. Any values submitted by voters in B_1 are simply ignored. The values that the honest voters created for the voters in B_1 will remain

unused, which is a problem since the numbers should sum to 0. To correct this, the honest voters open all encryptions assigned to voters in B_1.

More formally, for all $i \in R \backslash B_1$ the voter i releases the values s_{ij}, r_{ij} for all $j \in B_1$. Since these values are uniquely determined by the encryption, this step can be verified by checking that $c_{ij} = \widehat{E}^t_{(s),pk_j}(s_{ij}, r_{ij})$. Should a voter refuse to publish this information he is simply added to B_1 and his values are revealed.

Correctness and Security of the Setup Phase

Now we can prove several properties about the initialization phase that are useful for proving the correctness and security of the complete protocol.

Lemma 4.2 *The error correction step can always be completed if there are honest voters. After the completion of the initialization step and the error correction of this, the following will hold:*

1. *Any third party can verify that c_{ij} is a correct encryption for any $i \in R \backslash B_1$ and $j \in R_0$.*
2. *Any third party can verify that $\sum_{j \in R_0} s_{ij} = 0 \bmod n^s$ for any $i \in R \backslash B_1$.*
3. *If any voter $i \in R \backslash B_1$ chooses the value s_{ij} at random, the value $t_j = \sum_{i \in R \backslash B_1} s_{ij}$ is a random element in $\mathbb{Z}_{n^s}$ and $\sum_{j \in R_0} t_j = 0 \bmod n^s$.*
4. *Any third party can verify that the value s_{ij} released during error correction is indeed the plaintext inside c_{ij} for all $i \in R \backslash B_1$ and $j \in B_1$.*

Proof.

The error correction phase only requires the voter to show that he knows the values inside the ciphertexts. So, if there are any honest voters, this process will terminate at some point, and the honest voters can always provide all the values needed to complete this error correction step.
1) This is an implication of lemma 2.8, and the fact that voters submitting bad proofs are added to B_1 during the error correction.
2) The proof performed shows that the value $H_{i0} \cdots H_{iu}$ is a n^s'th power. This is true iff $\sum_{j \in R_0} s_{ij} = 0 \bmod n^s$ and it follows from lemma 2.3 that the proof has the desired effect.
3) Given a voter k that chose the element at random we have that:

$$t_j = s_{kj} + \sum_{i \in R \backslash (B_1 \cup \{k\})} s_{ij}$$

the value $\sum_{i \in R \backslash (B_1 \cup \{k\})} s_{ij}$ can be seen as some fixed element in $\mathbb{Z}_{n^s}$. Now since s_{kj} is chosen at random in $\mathbb{Z}_{n^s}$, the element t_j can be any element in $\mathbb{Z}_{n^s}$. This implies, that given a random s_{ij}, the value t_j can be seen as a random element in $\mathbb{Z}_{n^s}$.
4) No power of h can make any contribution to $(n+1)^{s_{ij}}$. This means that if an incorrect s'_{ij} is released, there will be no chance of providing a value r'_{ij} such that

$$c_{ij} = \widehat{E}^t_{(s),pk_j}(s'_{ij}, r'_{ij})$$

□

4.4.2 Ballot Casting

Ballot Casting

Each voter $j \in R \backslash B_1$ retrieves the encryptions c_{ij} for all $i \in R \backslash B_1$ and combines them:

$$c_j = \prod_{i \in R \backslash B_1} c_{ij} = \widehat{E}^t_{(s),pk_j}(\sum_{i \in R \backslash B_1} s_{ij}, r)$$

for some value of r. Voter j decrypts c_j using the private key α_j to get $t_j = \sum_{i \in R \backslash B_1} s_{ij}$.

Voter j then submits the values $d_j = \widehat{E}^t_{(s),pk_j}(v_j, r_j)$ and $x_j = v_j + t_j$, where v_j is the value representing the candidate that voter j votes for, say 0 or 1 for a yes/no election or M^{v_j} in a multi candidate election (where $M = u + 1$); the value $r_j \in \mathbb{Z}_\theta$ is chosen at random. The easiest way to understand this is to note, that if we ignore the error correction (i.e. assume that no faults occur), then the t_j's will be a set of random numbers that sum to 0. So, if we can ensure that x_j was formed by adding an allowable value of v_j to t_j, then $res = \sum_j x_j$ will be the election result, i.e. the sum of the v_j's. Moreover, the randomness of the t_j ensures that given the x_j's, all possible sets of v_j's summing to res are equally likely.

To prove that d_j is a legal encryption of an allowable value of v_j, the proof from section 3.3 can be used to ensure that it is a correct encryption, and the proof of a legal vote value (which is logarithmic in the number of candidates) from section 4.2.4 can be used on the second value in the encryption to prove that a legal v_j have been encrypted. To prove that d_j is an encryption of the same v_j, which was used to make x_j, the voter proves that

$$d_j c_j \widehat{E}^t_{(s),pk_j}(x_j, 0)^{-1} = \widehat{E}^t_{(s),pk_j}(0, r')$$

for some given r' using the n^s'th power proof from section 2.4.2 and sends this proof to the bulletin board. This time the required n^s'th root can be computed as: $h_j^{r_j} \cdot (\prod G_{ij}^{4\Delta^2 \alpha_j}) \cdot 1$. This holds iff the same v_j is used in d_j and x_j since

$$\begin{aligned} d_j c_j (\widehat{E}^t_{(s),pk_j}(v_j + t_j, 0))^{-1} &= \widehat{E}^t_{(s),pk_j}(v_j, r_j) \widehat{E}^t_{(s),pk_j}(t_j, r) \widehat{E}^t_{(s),pk_j}(-(x_j), 0) \\ &= \widehat{E}^t_{(s),pk_j}(v_j + t_j - (v_j + t_j), r') \\ &= \widehat{E}^t_{(s),pk_j}(0, r') \end{aligned}$$

Error Correction of Ballot Casting

Let B_2 be the set of voters disqualified during the ballot casting. Again there are some values that will not be used by the voters in B_2, and these are simply published on the bulletin board as in the error correction of the initialization.

However, this time the values created by voters in B_2 have been used by the honest voters (for any $i \in B_2$ the value of s_{ii} is only know by i, and all honest voters j have used s_{ij}). To correct this, the values have to be published, but the secret values r_{ij} used in c_{ij} are unknown to voter j. So, for all $i \in B_2$ each

$j \in R_0 \backslash (B_1 \cup B_2)$ (voters and bulletin board) decrypts and reveals the plaintext of c_{ij}, which is s_{ij}, and proves that

$$c_{ij} \widehat{E}^t_{(s),pk_j}(s_{ij}, 0)^{-1} = \widehat{E}^t_{(s),pk_j}(0, r)$$

using the n^s'th power proof from section 2.4.2. This time the required n^s'th root is: $G_{ij}^{4\Delta^2 \alpha_j} \cdot 1$. Should anyone refuse to participate in the error correction, they are simply added to B_2 and their values published as before.

Now let $B_{bad} = B_1 \cup B_2$ denote all voters that have been removed in the error correction steps, and let $R_{good} = R \backslash B_{bad}$ be the voters that completed the whole protocol honestly.

Post Ballot Casting

When the ballot phase is over, and all parties have either submitted their vote or been removed using the error correction, the bulletin board computes

$$c_0 = \prod_{i \in R \backslash B_1} c_{i0} = \widehat{E}^t_{(s),pk_0}(\sum_{i \in R \backslash B_1} s_{i0}, r)$$

for some r and gets the plaintext $t_0 = \sum_{i \in R \backslash B_1} s_{i0}$ by decrypting c_0. The bulletin board then posts t_0 along with a proof that the second element in $c_0 \widehat{E}^t_{(s),pk_0}(t_0, 0)^{-1}$ is an n^s'th power according to the proof in section 2.4.2 (the root is calculated as $(\prod G_{i0}^{4\Delta^2 \alpha_0}) \cdot 1$).

Correctness and Security of the Ballot Casting Phase

After the ballot casting phase and the error correction phase, we want to ensure that the voter did indeed submit the right value, and that no one can learn anything on v_j given x_j.

Lemma 4.3 *The error correction step can always be completed if there are honest voters. After the completion of the initialization step and the error correction afterwards, the following will hold:*

1. *Any third party can verify that a submitted vote x_j has the form $t_j + v_j$, where v_j is a legal vote and $t_j = \sum_{i \in R \backslash B_1} s_{ij}$.*

2. *The ballot x_j for $j \in R_{good}$ is equally likely for any legal vote if just one voter $i \in R_{good}$ chooses s_{ij} at random.*

3. *Any third party can verify that the value s_{ij} released during the first part of the error correction is indeed the plaintext inside c_{ij} for all $i \in R_{good}$ and $j \in B_2$.*

4. *Any third party can verify that the value s_{ij} released during the second part of the error correction is indeed the plaintext inside c_{ij} for all $i \in B_2$ and $j \in (R \cup \{0\}) \backslash B_{bad}$.*

Proof. The first values released in the error correction can always be revealed following the argument in the proof of lemma 4.2. The second values are values that are received by the voter j. The values revealed here are the messages in the encryptions c_{ij}, where $i \in B_2 \subset R \backslash B_1$ and $j \in R_{good}$. It follows from point 1 of lemma 4.2 that, for these values of i and j, the encryption c_{ij} is correct. This means that voter j can decrypt and find the value $G_{ij}^{4\Delta^2\alpha_j}$ it needs to complete the proof of correct disclosure. So, as long as there are honest voters, the error correction phase can always be completed.
1) The proofs presented in section 4.2.4 and 2.7.2 will ensure that the encryption d_j contains a legal encryption of a legal vote. The encryption

$$d_j c_j \widehat{E}^t_{(s),pk_j}(x_j, 0)^{-1} = \widehat{E}^t_{(s),pk_j}(0, r')$$

will have a second entry that is an n^s'th power iff x_j is the sum of the encryptions inside d_j and c_j. The value c_j is the combination of the encryptions that defined t_j and d_j contains a correct vote v_j. So, x_j will have to be the sum of t_j and v_j.
2) This follows from lemma 4.2, point 3 that implies t_j is a random value. Given x_j this means that any value $x_j - v_j$ for all legal votes v_j are equally likely, so x_j cannot be used to distinguish which v_j was chosen. Furthermore, the encryptions d_j and c_{ij}'s do not reveal any information due to the semantic security of the cryptosystem.
3) This follows the exact same argument as point 4 in lemma 4.2.
4) This follows from the fact that the value

$$c_{ij} \widehat{E}^t_{(s),pk_j}(s'_{ij}, 0)^{-1}$$

is an n^s'th power iff $s_{ij} = s'_{ij}$. So, if an incorrect value has been released the proof cannot be completed according to lemma 2.3.

□

We also need to prove some properties on the correctness and fairness of the Post Ballot Casting:

Lemma 4.4 *The following properties hold*

1. *Without the value t_0 a voter doesn't know anything about the outcome if just one voter i chooses s_{i0} at random.*
2. *Any third party can verify that the value t_0 released by the bulletin board is the value inside c_0.*

Proof.
1) Follows directly from the semantic security of the cryptosystem (i.e. no one can guess t_0 from the encryption) and from point 3 of lemma 4.2.
2) This is an immediate consequence of the correctness of the n^s'th power proof, which is proven correct in lemma 2.3, and the fact that c_0 is the combination of legal encryptions.

□

4.4.3 Tallying

At this point the result can be computed as:

$$\begin{aligned} res &= t_0 + \sum_{j \in R_{good}} x_j + \sum_{i \in R_{good}, j \in B_{bad}} s_{ij} - \sum_{i \in B_2, j \in R_{good} \cup \{0\}} s_{ij} \\ &= \sum_{j \in R_{good}} v_i \bmod n^s \end{aligned}$$

The first sum is all the x_j values that have been posted on the bulletin board according to protocol. The values in the second sum are the values of the disqualified voters, that were revealed in the error correction of the initialization and the first value revealed in the error correction of the ballot casting. The third sum is the sum of the second values revealed in the error correction of the ballot casting. Now we can prove the following result:

Theorem 4.2 *The protocol described above is a voting system that satisfies correctness, privacy, fairness, universal verifiability, self-tallying, dispute-freeness, perfect ballot secrecy, and corrective fault-tolerance.*

Proof.

Correctness: We have that R_{good}, $\{0\}$, B_1 and B_2 are disjoint and that

$$B_{bad} = B_1 \cup B_2 \text{ and } R = R_{good} \cup B_{bad} \text{ and } R_0 = R \cup \{0\}$$

Given point 3 of lemma 4.2 ($\sum_{j \in R_0} t_j = 0 \bmod n^s$) we get that

$$\begin{aligned} & t_0 + \sum_{j \in R_{good}} x_j \\ &= \sum_{j \in R_{good}} v_j + \sum_{j \in R_{good} \cup \{0\}} t_j \\ &= \sum_{j \in R_{good}} v_j - \sum_{j \in B_{bad}} t_j \bmod n^s \end{aligned}$$

Using the way t_j is defined we get

$$\begin{aligned} & t_0 + \sum_{j \in R_{good}} x_j + \sum_{i \in R_{good}, j \in B_{bad}} s_{ij} \\ &= \sum_{j \in R_{good}} v_j - \sum_{j \in B_{bad}} t_j + \sum_{i \in R_{good}, j \in B_{bad}} s_{ij} \\ &= \sum_{j \in R_{good}} v_j - \sum_{i \in B_2, j \in B_{bad}} s_{ij} \bmod n^s \end{aligned}$$

Using point 2 from lemma 4.2 ($\sum_{j \in R_0} s_{ij} = 0 \bmod n^s$ for any $i \in R \backslash B_1$) and that $B_2 \subset R \backslash B_1$, this immediately implies the correctness of the

result

$$\begin{aligned} res &= t_0 + \sum_{j \in R_{good}} x_j + \sum_{i \in R_{good}, j \in B_{bad}} s_{ij} - \sum_{i \in B_2, j \in R_{good} \cup \{0\}} s_{ij} \\ &= \sum_{j \in R_{good}} v_j - \sum_{i \in B_2, j \in R_0} s_{ij} \\ &= \sum_{j \in R_{good}} v_j \bmod n^s \end{aligned}$$

as long as the sum of v_j's is less than n^s.

Privacy: follows directly from point 2 in lemma 4.3, which implies the vote is hidden.

Fairness: direct consequence of lemma 4.4 point 1.

Universal Verifiability: comes from using the bulletin board and that every step of the protocol is verifiable (lemma 4.2 (1, 2, 4), lemma 4.3 (1, 3, 4) and lemma 4.4 (2)).

Self-Tallying: this follows from the correctness and that the result is computed using only values posted on the bulletin board by the voters.

Dispute-Freeness: all communication goes through the bulletin board, and all encryptions are proved to be legal. This means that no dispute can arise due to bad encryptions, and the values posted to the bulletin board can be verified by anyone.

Perfect Ballot Secrecy: we can ignore those encryptions that remain unopened, due to the semantic security of the cryptosystem. What remains are the public numbers: x_j's, and the numbers revealed during error correction. Now for any given subset $Y \subset R_{good}$ we have, if just one voter i in $R_{good} \backslash Y$ chooses s_{ij} at random the value t_j will be random from the view of Y. This means that the vote v_j cannot be guessed given just x_j.

Corrective Fault-Tolerance: lemma 4.2 and 4.3 imply that the error correction phase can always be performed, so we can always correct any faults that occur.

□

Remark 4.1 *The bulletin board does not have to participate in the error correction phase of the ballot casting. This can be achieved by making the bulletin board release the value t'_0 that is inside:*

$$c'_0 = \prod_{i \in R \backslash B_{bad}} c_{i0} = \widehat{E}^t_{(s),pk_0}\Big(\sum_{i \in R \backslash B_{bad}} s_{i0}, r\Big)$$

This means that

$$t_0 = t'_0 + \sum_{i \in R \backslash B_2} s_{i0} \bmod n^s$$

To get the result, the bulletin board has already removed the values from the bad votes in B_2 *and the result can be computed as:*

$$\begin{aligned} res &= t'_0 + \sum_{j \in R_{good}} x_j + \sum_{i \in R_{good}, j \in B_{bad}} s_{ij} - \sum_{i \in B_2, j \in R_{good}} s_{ij} \\ &= \sum_{j \in R_{good}} v_i \bmod n^s \end{aligned}$$

4.4.4 Efficiency Comparison to Scheme from [47]

For the case of a yes/no election, the work of the 2 schemes are comparable in all steps of the protocol except in the tallying phase. Here the protocol of [47] needs to do an exhaustive search in a space of size $2u$, which can be optimized to $\mathcal{O}(\sqrt{u})$ multiplications. However, the protocol above obtains the result of the election by simply adding the values posted to the bulletin board.

This scheme generalizes to multi-candidate elections in exactly the same way as in section 4.2.3. In particular, the tallying phase remains at the same number of additions. For the scheme from [47], the search for the result would take $\Omega((\sqrt{u})^L)$ multiplications for L candidates. Furthermore, the proofs of correctness used in the ballot casting phase will have size $\mathcal{O}(log(L))$ as opposed to $\mathcal{O}(L)$ in [47].

Chapter 5

Key Escrow

I appreciate their willingness to make some of that crypto research available to a public which has paid so much for it, but I'm afraid that I would never trust an algorithm which was given to me by any government.
— John Perry Barlow - On-line debate on Clipper

In chapter 2 we saw two kinds of cryptosystems. The El Gamal variant assumed that a modulus with unknown factorization was set up in advanced. However, if anyone knows the factorization they are able to decrypt encryptions from the non El Gamal variant in chapter 2. The second entry in the encryption of the El Gamal variant is exactly such an encryption. This means that a person who knows the factorization of the modulus can decrypt any ciphertext that was encrypted using a public key generated in the El Gamal variant based on this modulus. Using this we can create a key escrow system by having the escrow authority know the factorization of the modulus. Then anyone can create their own key using this modulus. This is a significant improvement to previous proposals, where the escrow authority has to hold one piece of information for every key used in the system.

5.1 Introduction

5.1.1 Background

During the last decade there has been a large growth in communication over the Internet. This has lead to an increased focus on privacy and other forms of security. However, sending messages encrypted poses a problem for law enforcement agencies (LEAs), which have relied on their ability to make wiretaps to solve crimes. With encrypted communication the LEA will be unable to monitor communication.

This has led to several key escrow proposals, in which the persons communicating will reveal their keys (or part of these) to the LEAs. This enables the LEA to decrypt messages, but poses the problem that it can also decrypt messages that it is not supposed to. Two ways have been proposed to combat this problem: 1) partial key escrow where the LEA only receives part of the key and has to do an exhaustive search to find the rest, and 2) introducing a Key

Escrow Agency (KEA), that handles the secret keys and helps the LEAs in the case of a lawful request. Partial key escrow relies on the assumption that the LEA cannot do a massive decryption of messages because of the work involved in finding the last portion of the key. However, this does not prevent a LEA to pick a few number of persons that it is not entitled to monitor and decrypt their communication. In both cases, there is a problem if the escrowed keys are reused. This will enable the LEA to keep monitoring the communication after the duration of the warrant has expired. However, this can be fixed by only escrowing session keys which are typically short lived.

In [55] Shamir proposed a scheme with partial key escrow. The idea behind partial key escrow is that the user escrows (reveals) for instance 8 bit of a DES key to the LEA. If the LEA at some point want to wiretap the user they will have to do an exhaustive search on the last 48 bits. This means that recovering a single key is cumbersome, but definitely possible, whereas recovering a lot of keys at the same time is hard due to the work load.

Concurrently to the result of Shamir, a similar idea was proposed by Micali in [49], which handled partial escrow of public keys. The schemes in [55] and [49] were later merged into a joint paper [50].

In [11] Bellare and Goldwasser proposed a verifiable partial key escrow scheme, which makes it possible for the receiver to check that the sender has escrowed the correct bits and not some random bits. If random bits where used the escrow agency would not have any help in guessing the key, and it would become infeasible to find it. They also address problems from [49] with early recovery, which means that the LEA is able to do the computation before receiving the key escrow information and thus get the key quickly upon receiving the escrow information.

Mao introduced a scheme in [48] where escrowed values could be publicly verified. This has the advantage that the escrowing authorities, the senders, and other interested parties can verify that encryptions are indeed subject to escrow.

A scheme proposed by Shaoquan and Yufeng [56] used a different setup, in which the LEA does not hold the escrow values. Instead a KEA holds shares of the escrowed values and discloses these to the LEA upon request. This setup is more like the existing power structures, where the different LEAs are independent from the judiciary system. The usual way to get a search warrant is that the LEA presents its case to a judge, who makes the decision whether to allow the search or not. It seems logical that the same should hold for electronic information, in which case the KEA should be a separate unit under e.g. the department of justice.

5.1.2 Contribution

We introduce a new cryptosystem that has two kinds of secret keys. First, there are several normal keys as introduced in section 2.5. Secondly, there is a global master key that is able to decrypt any message encrypted with the normal keys. This master key is a key similar to the keys for the generalized Paillier defined in section 2.2. Using the master key to reveal the decryption information will not

reveal any information on the normal private key that could also have decrypted the message.

This can be used to make key escrow by having the global master key shared between the escrow servers in a threshold fashion. Note, that since there is just one master key, the escrow servers do not have to keep a secret sharing of the secret keys of all the different users, as opposed to all other schemes to date.

The setup used in this paper is a setup similar to [56], since we have: 1) some users sending encrypted messages to each other, 2) a Key Escrow Agency (KEA) holding the escrow key, and 3) the Law Enforcement Agencies (LEAs) being agencies like FBI, CIA, county sheriff department, etc.

For a LEA to decrypt a message it will ask the KEA servers to provide a decryption value. The KEA generates the random value used in the encryption and sends it privately to the LEA. The LEA can then remove the random part from the encryption and get the message from the resulting value.

The system has the added advantage over existing protocols, that the users do not have to perform an expensive key escrow protocol with the KEA (or LEA) when setting up the system. The KEA simply generates some global parameters, and all the users generate a key pair in this global setup. This allows the KEA to "decrypt" without even knowing the public key of the user.

5.2 Model

The model consists of three kinds of players: 1) the users, 2) the KEA servers and 3) the LEAs.

The adversary model is two-sided. The players want to try and cheat the LEA so they are not able to decrypt their messages, and the LEAs try to decrypt messages they are not supposed to. The KEA works as a buffer between the two by providing decryptions to the LEA when it gets a valid request, and refuse when the LEA is trying to cheat.

The users are assumed to be able to mount several attacks, namely: 1) flooding: the user floods the channels with a lot of illegal encryptions and one legal encryption to make the LEA waste a lot of resources, trying to find the single legal encryption, 2) collude with some of the KEA servers to make the KEA servers unable to make the decryption value for the LEA, and 3) decrypt messages of other users by colluding with some KEA servers.

We will assume a LEA adversary that can control up to t KEA servers and a number of users (informants). The goal of this adversary is to achieve one of the following: 1) find the decryption of a ciphertext, 2) find the private key of a user, or 3) find the secret shared between the KEA servers. The third attack is the most dangerous attack, since it will enable the LEA to decrypt all messages, without help from the KEA servers at all.

For the threshold version of the KEA servers we will assume, that the KEA servers have a bulletin board they can access to make decryptions. This is used for distributing their decryption values during decryption and is not required to be kept secret.

5.3 A Simple Key Escrow System

The idea in this section is to let the KEA server set up the global values for the proof friendly cryptosystem $\widehat{CS}^t$ from section 2.7. This means that the KEA will know the factorization of n, which will allow it to compute the random value $h^r \bmod n$ (or $h^{4\Delta^2 r} \bmod n$ depending on which system is used), used in the encryption with the technique from theorem 2.1. This is done by iterating the technique from theorem 2.1 s times using n instead of n^s. The reason for this is to allow an easier transition to a threshold system.

In section 5.4, the simple system is changed into a threshold system with several KEAs and verification between the KEA servers and the LEA. The problem of users trying to flood LEAs are addressed in section 5.5 by adding proofs of legal encryption to the encryption step. In most cases $s = 1$ will be sufficient for the key escrow scenario, but for completeness we will describe it for a general s, and in section 5.6 we will address some of the efficiency issues arising from using larger s values.

Global Setup (KEA):

1. Pick 2 primes p, q of size $k/2$ bits each, where k is the security parameter. They should also satisfy that $p = 2p' + 1$ and $q = 2q' + 1$ for primes p', q' (i.e. p and q are safe primes).
2. Set $n = pq$ and $\tau = p'q'$.
3. Pick $g \in \mathrm{Q}_n$, the group of squares.
4. Release the parameters: (n, g).
5. Store the escrow key: $d = n^{-1} \bmod \tau$.

Key Generation (user i):

1. Pick $\alpha_i \in \mathbb{Z}_\theta$, where θ is the value introduced in section 2.3.1.
2. Set $h_i = g^{\alpha_i} \bmod n$.
3. Release the public key: $pk_i = (n, g, h_i)$.
4. Store the secret key: α_i.

Encryption (using pk_i):
To encrypt message $m \in \mathbb{Z}_{n^s}$, choose $r \in \mathbb{Z}_\theta$ and $b_0, b_1 \in \{0, 1\}$:

$$\begin{aligned}\widehat{E}^t_{pk_i}(m, r, b_0, b_1) &= (G, H)\\ &= ((-1)^{b_0} g^r \bmod n,\\ &\qquad (-1)^{b_1}(h_i^r \bmod n)^{n^s}(n+1)^m \bmod n^{s+1}\end{aligned}$$

Note that this is not the exact same function as in section 2.7. In general the escrow will work if the encryption is of the form

$$((-1)^{b_0} g^r \bmod n, (-1)^{b_1}(h_i^{\beta r} \bmod n)^{n^s}(n+1)^m \bmod n^{s+1})$$

where β is some fixed value. In section 2.7 the value of β was $4\Delta^2$.

Decryption (using α_i):
To decrypt (G, H):

$$m = \mathrm{dLog}_s((G^{\alpha_i} \bmod n)^{-2n^s} H^2 \bmod n^{s+1})/2 \bmod n^s$$

Key Escrow (KEA):
Given (G, H):

1. Abort if either G or H is malformed (i.e. $\gcd(G, n) \neq 1$, $\gcd(H, n) \neq 1$ or either G or H has Jacobi symbol -1 wrt. n).
2. Compute: $x_s = H \bmod n = \pm(h_i^r)^{n^s} \bmod n$.
3. Compute x_0 using s repetitions of:

$$x_{k-1} = (x_k)^d = (\pm(h_i^r)^{n^k})^{n^{-1}} = \pm(h_i^r)^{n^k n^{-1}} = \pm(h_i^r)^{n^{k-1}} \bmod n$$

4. Send x_0 securely to the LEA.

Escrowed Decryption (LEA):
Given (G, H) and x_0, compute:

$$m = \mathrm{dLog}_s((x_0)^{-2n^s} H^2 \bmod n^{s+1})/2 \bmod n^s$$

The work of the KEA server is of the order $\mathcal{O}(sk^3)$, where k is the security parameter (size) of the modulus n, and the work of the LEA is $\mathcal{O}((sk)^3)$. If the server keeps τ instead of d it can generate $d' = n^{-s} \bmod \tau$. Using this it can compute x_0 directly:

$$x_0 = (x_s)^{d'}$$

The work used in this case is only $\mathcal{O}(sk^2 + k^3)$.

Note, that the key generation of the users can be made in a threshold way to create a threshold decryption key. The escrow decryption will still work, even if the cryptosystem is changed slightly to accommodate the threshold version from section 2.6.

The above scheme, however, only works in the passive case. In the active case there are a lot of problems. When the LEA cannot decrypt (that is $(x_0)^{-2n^s} H^2 \bmod n^{s+1}$ is not a power of $(n + 1)$) there is no way to tell if it is because the user submitted a bad encryption (see 5.5) or if the KEA gave it a wrong value.

5.4 Threshold Key Escrow

To make the system threshold, we have to share the decryption value d and set up some verification values. Furthermore, the decryption phase has to provide verification proofs, so that KEAs cannot cheat the LEA and ruin the decryption at some point.

The protocol uses a Trusted Third Party (TTP) to set up the protocol, but in section 5.4.1 we show how to setup the system without a TTP.

Global Setup (TTP):

1. Pick 2 primes p, q, such that $p = 2p' + 1$ and $q = 2q' + 1$ for primes p', q' (i.e. p and q are safe primes).
2. Set $n = pq$ and $\tau = p'q'$.
3. Pick $g, v \in \mathrm{Q}_n$, the group of squares.
4. Compute: $d = (4\Delta^2 n)^{-1} \bmod \tau$, where $\Delta = w!$ for w KEA servers.
5. Pick random $a_i \in \mathbb{Z}_\tau$ for $i \in \{1, \cdots, t\}$, where $t < w/2$ is the threshold of the system.
6. Set $a_0 = d$ and create the polynomial $f(x) = \sum_{i=0}^{t} a_i x^i \bmod \tau$ (Shamir secret sharing [54]).
7. Release the parameters: (n, g).
8. Send $d_j = f(j)$ to the j'th KEA server.
9. Calculate: $v_j = v^{d_j} \bmod n$, for $j \in \{1, \cdots, w\}$.
10. Release the verification values: $(v, v_1, \cdots, v_w)$.

Key Generation (user i): As above.

Encryption (using pk_i): As above.

Decryption (using α_i): As above.

Key Escrow (each KEA server):
Given (G, H) we do as above, except the method for calculating x_{k-1} is now a distributed protocol:

1. Given x_k, each server j computes: $x_k^j = x_k^{2\Delta d_j}$.
2. Server j makes a proof that:
$$\log_{x_k^{4\Delta}}((x_k^j)^2) = \log_v v_i$$
which is the same proof as in section 2.7.2.
3. Server j sends x_k^j and the proof to the KEA bulletin board.
4. The servers check the proofs of the submitted values and picks a qualified set S with legal proofs and computes:
$$x_{k-1} = \prod_{j \in S} (x_k^j)^{2\lambda_j^S} \bmod n$$
where λ_j^S is the slightly modified Lagrange coefficient:
$$\lambda_j^S = \Delta \prod_{i \in S \setminus \{j\}} \frac{-i}{j-i}$$
This means that:
$$x_{k-1} = \prod_{j \in S} (x_k)^{4\Delta d_j \lambda_j^S} = (x_k)^{4\Delta^2 f(0)} = (x_k)^{n^{-1} \bmod \tau} \bmod n$$
which is what we want.

The decryption share x_1^j, and the proof is not posted to the bulletin board by server j, but is sent directly to the LEA using a secure authenticated channel.

Escrowed Decryption (LEA):
The LEA checks the proofs on the bulletin board, checks the proofs of the shares x_1^j, which were sent to it, and picks a set S with correct proofs. It performs the Lagrange combination as in step 4 of the key escrow to get x_0, and m is computed using x_0 as it was done in section 5.3.

5.4.1 Removing the Trusted Third Party

In the protocol above we made use of a trusted third party to set up the global values, the secret sharing of d and the verification values. This can be done in a distributed fashion, so that the KEA servers can perform the setup themselves without affecting the security of the system.

To generate a product of safe primes the technique from [5] can be used. Although this is somewhat expensive this is an operation that only needs to be done once at the setup of the protocol. This generates an additive sharing of p and p' and tests for the primality of both (and likewise for q and q'). A secret sharing of the modulus n is then created from the sharing of p and q and then opened. To get a sharing of τ the sharing of p' and q' can be combined in the same way.

To generate the random values a_i, the servers can simply choose sufficiently large random numbers (about $k + k_2$ bits, where k_2 is e.g. 160 bits) using the technique for creating the prime candidates p', q'. Then the protocol for reducing a shared secret modulo a shared secret can be used to reduce the random number modulo τ to create a value in $\{0, \cdots, \tau\}$, which will be statistically close to a uniform value.

The values g and v can be generated by generating two random elements $y, y' \in \mathbb{Z}_n^*$, using e.g. commitments. The values can then be set to $g = y^2 \bmod n$ and $v = y'^2 \bmod n$, which are both in Q_n.

Now all values used in the setup phase are either public or secret shared, and we can compute the rest using the general computation framework of [5].

5.5 Encryption Verification

The escrowed decryption cannot distinguish legal encryptions from illegal encryptions, since it computes the randomness used for the second part of the encryption. This means that a malicious sender could generate a lot of encryptions on the form:

$$(G, H) = (r_1, (r_2)^{n^s}(n+1)^{m^*} \bmod n^{s+1})$$

The values r_2, m^* might not be known by the adversary, but 2 such values exists. The decryption by a user will show, that it is an illegal encryption and it will be discarded, whereas the escrow decryption will result in the value r_2 being passed to LEA and the message m^* being output. The above encryption

Scheme	$\sharp d$	$\sharp$ verification values	$\sharp$ exponentiations when $s \leq s'$	$\sharp$ exponentiations when $s > s'$
Only 1	1	$w+1$	s	s
All	s'	$s'w+1$	1	s/s'
2 powers	$log_2(s')$	$log_2(s')w+1$	$\sim log_2(s)/2$	$\sim s/s' + log_2(s')$

Figure 5.1: Different values when using upper bound s' to set up the system

cannot be distinguished from a normal encryption (by conjecture 2.2). So the KEA or LEA servers can be overloaded by sending just 1 correct encryption and a lot of illegal encryptions as above. The receiver will discard all the illegal encryptions and accept the single correct encryption, whereas the LEA will have a lot of plaintexts of which only one is actually received.

To take care of this the sender makes a proof of legal encryption according to the protocol in section 2.7.2. The proof is made non-interactive by using a hash function to create the challenge. The challenge is made non-malleable by including some identifier of the sender and receiver in the hash, such that substituting the sender or the receiver will invalidate the correctness of the proof. Note that the proof does not use the fact that it is $h^{4\Delta^2 r}$ and not h^r that is used, so it will work for any $h^{\beta r}$ where β is some fixed number.

5.6 Improving Performance for $s > 1$

The calculation of x_0 require s rounds of exponentiation in the previous schemes. In the case where many messages use an $s > 1$, it might be an advantage to decrease the number of needed exponentiations in the escrow part of the protocol. To do this, extra decryption values can be computed:

$$d_s := 4\Delta^2 n^{-s} \bmod \tau$$

This will allow the servers to remove s powers of n in a single exponentiation step. To be able to verify correctness each of these new decryption values require an extra set of verification values against which the exponentiations are verified.

When there are only a few number of s's that are frequently used, these values can be computed in advance together with the verification values. If the KEA servers keep the sharing of τ they can compute new values after the setup phase is done.

If s is arbitrary in general, there are different strategies that can be used to reduce space (number of values kept by the KEAs and size of all verification values), time and communication (number of rounds of exponentiation to compute x_0). In figure 5.1 three different approaches are given some upper bound s': 1) only d_1 is used, 2) use all the $d_1, d_2, \cdots, d_{s'}$, and 3) use only the powers of 2: $d_{2^0}, d_{2^1}, \cdots, d_{2^{log_2(s'/2)}}$.

5.7 Security of the System

Here is an informal argumentation about why this scheme is secure. This is based on the correctness of the different proofs used and the semantic security of the cryptosystem:

Theorem 5.1 *Senders cannot flood any LEAs.*

Proof sketch. This follows from the correctness of the proof of correct encryption. Since the sender has to create a correct proof it will be unable to fool a LEA into decrypting bad messages. □

Theorem 5.2 *Users cannot prevent decryption when controlling less than $w/2$ servers.*

Proof sketch. The exponentiation of the KEA servers uses proof of correct behavior, which means that the user cannot inject bad values without being noticed with all but a negligible chance.

The secret d is shared between w servers with a threshold of $t < w/2$. If the user controls less than $w/2$ servers there will be at least $t+1$ honest servers left which is enough to perform the exponentiation. This means that the servers will be able to finish the protocol and give the correct value to the LEA. □

Theorem 5.3 *A user cannot decrypt messages from other users when t or less KEA servers are helping.*

Proof sketch. This follows directly from the semantic security of the cryptosystem and the fact that t or less KEAs have no information on the shared secret. □

Theorem 5.4 *The LEA cannot decrypt messages encrypted by users without getting the decryption value from KEA.*

Proof sketch. This is the same as for theorem 5.3, except that LEAs can ask for getting messages decrypted. There are 2 reasons for granting such a decryption, namely if either sender or receiver is considered suspicious. Now, if the original sender/receiver pair is not considered suspicious, the LEA will have to create a related ciphertext where either the sender or receiver is a suspicious person.

However, if the sender or receiver is changed the input to the hash function is changed and another challenge will be used for the proof of a legal encryption. This means that the message cannot be changed to look like it is to/from some suspicious person. □

Theorem 5.5 *A LEA learns no non-trivial information on the private key of the user during decryption.*

Proof sketch. The signature scheme in [57] by Shoup is proven secure in the random oracle model. This means, that since each exponentiation step in the escrow protocol is exactly the same as a Shoup threshold signature computation, they are by themselves secure.

The different results after each exponentiation offer no information either, since such tuples can be generated by the adversary himself. This can be done by picking r and then computing $(g^r, h^r, (h^r)^n, \cdots, (h^r)^{n^s})$. □

Theorem 5.6 *Users, LEAs and t or less KEA servers cooperating are unable to calculate d.*

Proof sketch. Firstly t or less KEA servers have no information on d, since it is secret shared with a threshold of t.

Users can only get correctly constructed ciphertexts decrypted. This means that the values the KEA servers raise to the secret exponent d_j is on the form:

$$(h^r)^{\beta n^s} \bmod n$$

which is in Q_n. However, this is the exact same type of values that are exponentiated in [57]. If an adversary exists against this step then an adversary exist against Shoup's scheme, which was proven secure in the random oracle model.

The rest follows from the proof of theorem 5.5, namely that the rest of the values can be simulated by the LEAs themselves, without help from the KEA servers. □

Chapter 6

Efficient Petitions

In science it often happens that scientists say, 'You know that's a really good argument; my position is mistaken,' and then they actually change their minds and you never hear that old view from them again. They really do it. It doesn't happen as often as it should, because scientists are human and change is sometimes painful. But it happens every day. I cannot recall the last time something like that happened in politics or religion.
— Carl Sagan, 1987 CSICOP keynote address

In this chapter we look at petitions. These protocols can be used to collect signatures and hand them over to some verifier who verifies the signatures. It is simple to create such a protocol using any standard signature scheme, so in this chapter we look at some alternative ways to collect signatures. One of these greatly improves the performance compared to using standard signatures by combining several signatures into just one signature that the verifier has to check.

6.1 Introduction

6.1.1 Background

In [14] Boneh, Lynn, and Shacham proposed a signature scheme based on the Weil pairing with very short signatures (120-265 bits depending on the security parameter). The signature scheme is based on Gap Diffie Hellman groups (of which the Weil pairing is one). These are groups where the Computational Diffie Hellman problem (i.e. computing discrete logs) is assumed hard, but where the Decisional Diffie Hellman problem is easy.

6.1.2 Related Work

In work independent from, but concurrent to ours, Boldyreva proposed the notion of threshold signatures, aggregate signatures and blind signatures based on Gap Diffie Hellman groups. This result covers the basic building blocks in our scheme (threshold and aggregate signatures). The paper does not consider the application to petitions, which is presented in this chapter.

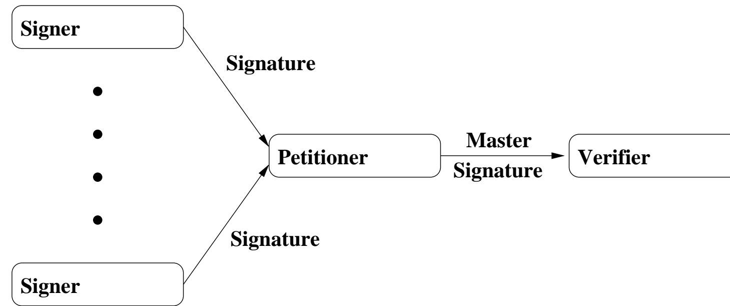


Figure 6.1: The general setup of a petition system.

6.1.3 Contribution

Petitions are a multi-party protocol consisting of 3 kinds of participants: A *Petitioner* who wants to convince a *Verifier*, that it has received a certain amount of signatures from different *Signers*.

In this chapter we will look at 3 different petition systems: 1) a system built using any normal signature system, 2) a system based on threshold signatures, and 3) a system using aggregate signatures. The first two systems follow directly from normal and threshold signature schemes, respectively. The third system, however, uses a new concept of aggregate signatures, which is presented here.

6.2 Model

The model consists of a *Petitioner* P, a *Verifier* V and some *Signers* $S_1, ..., S_w$. P has an *Issue* I that he collects signatures for. Once he has collected enough he presents them to V in order to try and persuade V to take some action. V checks the result from the petitioner to see if the petitioner has collected the claimed signatures.

The role of the petitioner is to collect the signatures and remove all illegal entries (invalid signatures and double signatures), such that the burden on the verifier is minimized. The reason for this is twofold: 1) the petitioner needs to keep track of how many signatures he has collected so far, and 2) the petitioner usually works on a few petitions whereas the verifier might be some government office that receives a lot of petitions. It is therefore more important to reduce the load placed on the verifier than on the petitioner. When all the signatures have been collected, a master signature is sent to the verifier. The structure of the master signature can be very different, ranging from a list of signatures (and the identities of the signers) to just a single signature. Given the master signature, the verifier checks this and accepts the parts of it that are correct. A sketch of this setup can be seen in figure 6.1.

So given this setup what can we hope to achieve in terms of communication and computational complexity?

Each signer has to generate a signature of some kind and send it to the petitioner. This means, that a lower bound on computation for the signer is

generating 1 signature, and the communication complexity is at least the size of a signature.

From the petitioner to the verifier, the best we can hope for is, that only one signature is sent. However, sending only one signature will require some sort of combination of all the signatures by the petitioner.

The verifier has to check the master signature, which is received from the petitioner. The best we can hope for is that the verifier only has to check one signature.

6.2.1 Properties for Petitions

One aspect of petitions is whether there is some threshold that needs to be achieved to convince V or whether it is a question about providing as many signatures as possible to appear "convincing", i.e. show that people care about the issue in question. This leads to 2 classifications:

Exact: In these the exact number of legal signatures will be known. This is useful for petitions that are used to convince politicians that an issue is important for a lot of people (e.g. save the whales).

At least: Here the exact number of signatures made is not known, but only that more than a certain threshold of signers have signed the petition. This can be used to show that a certain amount of people have signed the petition (e.g. getting a party accepted on the ballot in certain national elections).

6.3 Standard Petitions

Any signature scheme can be used in the obvious way. The signer creates a signature on $\mathcal{H}(I)$ and sends it to the petitioner. The petitioner collects all the signatures in a list along with the identity of the signer. When the petition is over P sends the list to V. V checks the petition result by verifying all the signatures in the list with the public key of the signers.

If n signatures have been submitted and n' of these are legal and unique, we get the following complexities:

Computation:

Each signer: 1 signature.

Petitioner: n verifications.

Verifier: n' verifications.

Total communication:

Each signer to Petitioner: 1 signature.

Petitioner to Verifier: n' signatures with the identity of the signer attached to each signature.

Storage space needed:

Petitioner: n' signatures.

Classification: Exact.

This is optimal for the signer, but the communication between the petitioners and the verifier is far from minimal. Furthermore, the verifier is burdened with checking all the signatures.

6.4 Threshold Petitions

Petitions can also be built using threshold signatures. The idea is that all signers have a share of the signing key. If a signer wants to sign a petition on I, it provides a signature share on $\mathcal{H}(I)$. When the petitioner has enough signatures, it combines the signature shares to get a signature on $\mathcal{H}(I)$. This signature is sent to the verifier, who checks the single signature.

This gives the following complexities, if t is the threshold of the signature system and n signature shares have been submitted before t legal proofs are found:

Computation:

Each signer: 1 signature share and a proof of correctness.

Petitioner: n proof checks and the Lagrange interpolation of t shares.

Verifier: 1 signature check.

Total communication:

Each signer to Petitioner: 1 signature share with a proof of correctness.

Petitioner to Verifier: 1 signature.

Storage space needed:

Petitioner: Up to t signature shares.

Classification: At least.

The work performed by the verifier and the communication from the petitioner to the verifier is minimal. The signer and the petitioner on the other hand have to do some more work, and the petition will only show that at least the required amount of people have signed the petition.

Note that, unlike the others systems in this chapter, it is problematic to include new signers. This essentially requires creating a new share of the secret each time a new signer is added. This is not trivial and at best a prohibitively expensive operation.

6.5 An Efficient Petition System

In this section a petition system is presented which uses the notion of aggregate signatures to greatly reduce the computational complexity of the verifier. Aggregate signatures are introduced in section 6.5.1 and used in section 6.5.5 to create a petition scheme that improves the standard signatures presented in section 6.3.

6.5.1 Aggregate Signatures

Aggregate signatures can be created given a signature scheme that supports the three normal functions *KeyGen*, *Sign*, and *Verify* and three aggregate functions *ComSign*, *ComPK*, and *RemSign*.

In aggregate signatures we are given two signatures $s_{sk_1}(m)$ and $s_{sk_2}(m)$ of the same message m, but under different keys. The goal is to create a signature $s_{f'(sk_1,sk_2)}(m)$, that can be verified using a combination $f(pk_1, pk_2)$ of the two public keys under which the original two signatures could be verified. For this to work, the combination of signatures (*ComSign*) and the combination of public keys (*ComPK*) needs to be efficiently computable.

There are 4 sets used in the following, namely: 1) secret keys ($\mathcal{SK}$), 2) public keys ($\mathcal{PK}$), 3) messages ($\mathcal{M}$) and 4) signatures ($\mathcal{S}$). Using these spaces we can define the 6 functions mentioned above as:

1. **Key generator:** $KeyGen : \{0,1\}^* \times \{0,1\}^* \rightarrow \mathcal{SK} \times \mathcal{PK}$.
   Given some global information and a random string, it creates a secret key sk and a public key pk that satisfies:

   $$Verify(Sign(m, sk), m, pk) = \text{ACCEPT} \quad \forall m \in \mathcal{M}$$

2. **Signature:** $Sign : \mathcal{M} \times \mathcal{SK} \rightarrow \mathcal{S}$.
   Taking a message and a secret key it provides a signature under the given secret key.

3. **Verify:** $Verify : \mathcal{S} \times \mathcal{M} \times \mathcal{PK} \rightarrow \{\text{ACCEPT}, \text{REJECT}\}$.
   Given a message, a signature and a public key, it returns whether the signature is a correct signature of the message given the public key.

4. **Combine PK:** $ComPK : \mathcal{PK} \times \mathcal{PK} \rightarrow \mathcal{PK}$.
   Combines 2 public keys into a new public key.

5. **Combine signature:** $ComSign : \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$.
   Combines 2 signatures s_1 and s_2 on the message m into a new signature s on the same message m. The function should satisfy:

   $$Verify(ComSign(s_1, s_2), m, ComPK(pk_1, pk_2)) = \text{ACCEPT}$$

   The function *ComSign* has to be associative and commutative.

6. **Remove signature:** *RemSign* : $\mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$.
   This function takes 2 signatures, where the first signature is of the form *ComSign*(s_1, s_2) and the second signature is s_2. The result of applying this function should be the signature s_1. The reason for this function is in the security analysis. The associative and commutative requirement for *ComSign*, allows *RemSign* to remove any signature from the aggregate signature and not just the last signature.

The following notation will be used when having several application of the same function:

$$f(x_1, x_2, x_3, ..., x_n) := f(...f(f(x_1, x_2), x_3)..., x_n)$$

6.5.2 Using Aggregating Signatures

The idea is, that given a list of signatures $(s_{i_1}, s_{i_2}, .., s_{i_n})$ on the same message m, a single signature s can be made instead. To be able to check this signature, the public keys corresponding to the original signatures are combined into a single public key, that can be used to verify s. So in general an aggregate signature will have the form (m, s, X), meaning all members of the set X have signed the message m and the combined signature is s. X has to be a multi-set, or signatures with $X_1 \bigcap X_2 \neq \emptyset$ cannot be joined without removing a signature representing $X_1 \bigcap X_2$ first. The basic ideas of the aggregate signatures can be summarized as follows:

Setup: Fix the global parameters if any, and then each participant can use the *KeyGen* algorithm to create a public/secret key pair. Each user is also assigned a public identifier, so public keys can be distinguished using this identifier.

Signing: The user i signs a message m by calculating:

$$s_{i,m} := Sign(m, sk_i)$$

The signature is $(m, s_{i,m}, \{i\})$ meaning the message m is signed by i.

Combining: Given 2 signature pairs on the same message: $(m, s_{X_1,m}, X_1)$ and $(m, s_{X_2,m}, X_2)$, a new signature can be created using

$$s_{X,m} := ComSign(s_{X_1,m}, s_{X_2,m})$$

where $X = X_1 \bigcup X_2$ is a multi-set union. The new signature tuple is $(m, S_{X,m}, X)$. This works because the *ComSign* function is assumed to be associative and commutative. The two signatures should be tested before the combination, or one could end up with an illegal signature, thereby loosing information about legal signatures.

Verify: Given $(m, S_{X,m}, X)$, where $X = \{i_1, ..., i_n\}$, verification can be done by computing

$$pk = ComPK(pk_{i_1}, pk_{i_2}, ..., pk_{i_n})$$

and then performing the test

$$Verify(S_{X,m}, m, pk)$$

6.5.3 Security of Aggregate Signatures

For analyzing the security it is assumed that the final signature is on some fixed set X ($n = |X|$) for which the adversary has to create a signature of a message m of his choice. Security of normal signatures and aggregate signatures can be defined as follows.

Definition 6.1 *A forger algorithm $\mathcal{F}$ (t, q, ϵ)-breaks a signature scheme if $\mathcal{F}$ runs in time t, makes q queries to a signing oracle S and is able to forge a signature s on a message m (for which it has not received a signature) with probability higher than ϵ (here r is the seed for key generation):*

$$\Pr\left[\mathit{Verify}(m, s, pk) = \mathit{ACCEPT} \;\middle|\; \begin{array}{c} (pk, sk) := \mathit{KeyGen}(r) \\ (m, s) := \mathcal{F}^S(pk) \end{array} \right] \geq \epsilon$$

A signature scheme is (t, q, ϵ)-secure against existential forgery by an adaptive chosen-message attack if no forger (t, q, ϵ)-breaks it.

In aggregate signatures we also have a set X of signers in the aggregate signature. So defining an adversary against an aggregate signature scheme, we will allow it to ask for signatures on the message it outputs, as long as it does not ask for signatures for all $i \in X$.

Definition 6.2 *A forger algorithm $\mathcal{F}$ (t, n, q, ϵ)-breaks an aggregate signature scheme if $\mathcal{F}$ runs in time t, makes q queries (m', i) to a signing oracle S getting $\mathit{Sign}(m', sk_i)$, and is able to forge a signature (m, s, X) with probability higher than ϵ for which there exists an $i \in X$, such that $\mathcal{F}$ has not asked for (m, i) from S (here r is the seed for key generation and g is the global information):*

$$\Pr\left[\mathit{Verify}(m, s, pk_X) = \mathit{ACCEPT} \;\middle|\; \begin{array}{c} \forall i : (pk_i, sk_i) := \mathit{KeyGen}(g, r_i) \\ (m, s, X) := \mathcal{F}^S(pk_1, ..., pk_n) \end{array} \right] \geq \epsilon$$

An aggregate signature scheme is (t, n, q, ϵ)-secure against existential forgery by an adaptive chosen-message attack if no forger (t, n, q, ϵ)-breaks it.

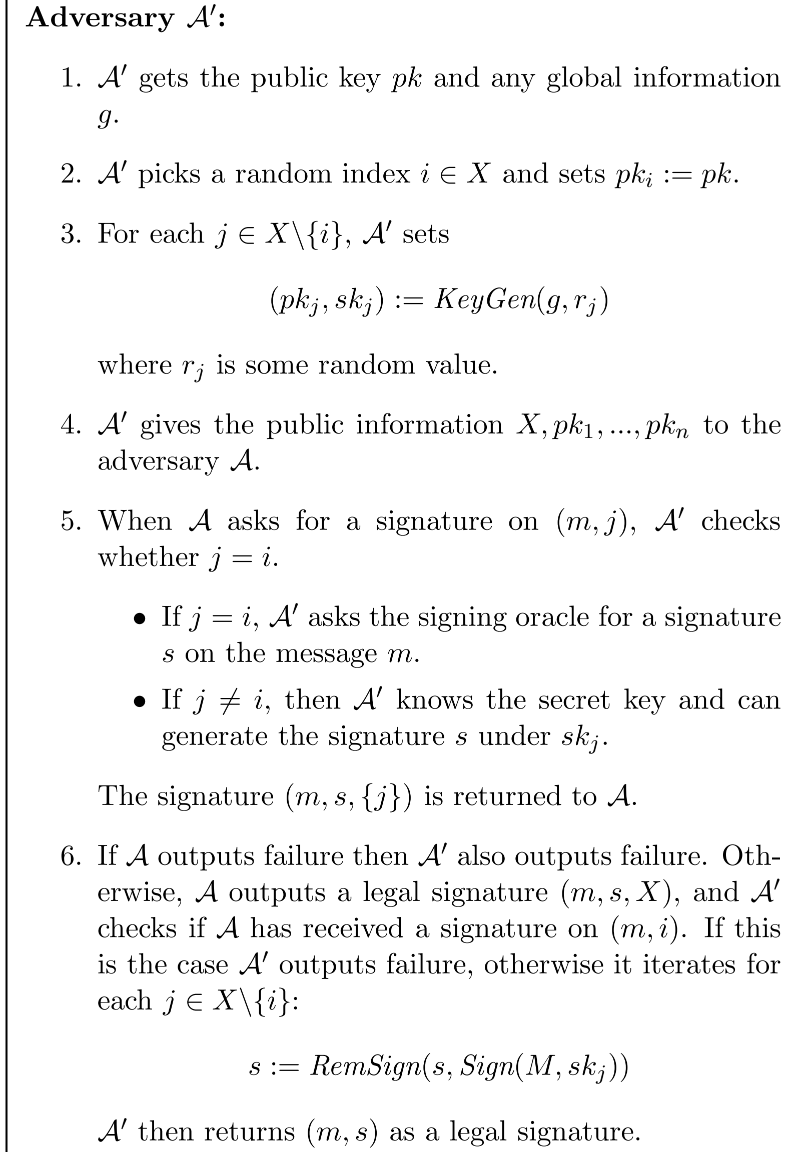
Using c_{KeyGen}, c_{Sign}, and $c_{RemSign}$ to represent the running time of *KeyGen*, *Sign*, and *RemSign* respectively, we can prove the following theorem:

Theorem 6.1 *Given an underlying signature scheme that is (t', q', ϵ')-secure, aggregate signatures based on this will be (t, n, q, ϵ)-secure against existential forgery by an adaptive chosen-message attack, where*

$$q \geq q' \quad \text{and} \quad \epsilon > n\epsilon' \quad \text{and}$$

$$t \geq t' - ((n-1) * (c_{KeyGen} + c_{RemSign}) + (q + n - 1) * c_{Sign})$$

Proof. Given an adversary $\mathcal{A}$ against the aggregate signature scheme we can construct an adversary $\mathcal{A}'$ against the underlying signature scheme. The adversary $\mathcal{A}'$ is shown in figure 6.2.



Adversary $\mathcal{A}'$:

1. $\mathcal{A}'$ gets the public key pk and any global information g.
2. $\mathcal{A}'$ picks a random index $i \in X$ and sets $pk_i := pk$.
3. For each $j \in X\backslash\{i\}$, $\mathcal{A}'$ sets

$$(pk_j, sk_j) := \mathit{KeyGen}(g, r_j)$$

   where r_j is some random value.
4. $\mathcal{A}'$ gives the public information $X, pk_1, ..., pk_n$ to the adversary $\mathcal{A}$.
5. When $\mathcal{A}$ asks for a signature on (m, j), $\mathcal{A}'$ checks whether $j = i$.
   - If $j = i$, $\mathcal{A}'$ asks the signing oracle for a signature s on the message m.
   - If $j \neq i$, then $\mathcal{A}'$ knows the secret key and can generate the signature s under sk_j.

   The signature $(m, s, \{j\})$ is returned to $\mathcal{A}$.
6. If $\mathcal{A}$ outputs failure then $\mathcal{A}'$ also outputs failure. Otherwise, $\mathcal{A}$ outputs a legal signature (m, s, X), and $\mathcal{A}'$ checks if $\mathcal{A}$ has received a signature on (m, i). If this is the case $\mathcal{A}'$ outputs failure, otherwise it iterates for each $j \in X\backslash\{i\}$:

$$s := \mathit{RemSign}(s, \mathit{Sign}(M, sk_j))$$

   $\mathcal{A}'$ then returns (m, s) as a legal signature.

Figure 6.2: Algorithm for adversary $\mathcal{A}'$ breaking the underlying signature system given an adversary $\mathcal{A}$ against the aggregate signature system.

It follows from the associative and commutative properties of *ComSign* and the fact that *RemSign* removes the signatures one by one, that the signature generated in step 6 will be a signature verifiable under the public key pk.

The chance for success is equal to that of the original scheme times the chance that the signatures $\mathcal{A}$ has not asked for is of the form (m, i). Since all the public keys are indistinguishable the chance that $j = i$ in step 5 is at least the chance of picking a random element from X. The probability of this is $1/n$ and thus the error probability is

$$\epsilon > n\epsilon'$$

The number of queries made to the signing oracle is the number of times $\mathcal{A}'$ has been asked to sign a message with user i's private key. This means that $\mathcal{A}'$ uses at most the same number of queries to the oracle as $\mathcal{A}$.

Running the adversary algorithm $\mathcal{A}'$ takes time equal to running $\mathcal{A}$ plus the time needed for computations. To set up the extra keys $\mathcal{A}'$ needs to do $n-1$ calls to *KeyGen*. Up to q calls to *Sign* are made to generate signatures in the cases where $j \neq i$. For generating the output signature the adversary $\mathcal{A}'$ needs to do $(n-1)$ calls to *Sign* and *RemSign*. This means that $\mathcal{A}'$ takes time at most:

$$t' \leq t + ((n-1) * (c_{KeyGen} + c_{RemSign}) + (q+n-1) * c_{Sign})$$

Conversely, if there does not exist an algorithm that (t', q', ϵ')-breaks the normal signature scheme there cannot exist an algorithm that (t, n, q, ϵ)-*breaks* the aggregate signature scheme where

$$t \geq t' - ((n-1) * (c_{KeyGen} + c_{RemSign}) + (q+n-1) * c_{Sign})$$

This concludes the proof. □

This also holds for signatures on a subset of the given set X, since the signature has to contain at least one signature $(m, s, \{j\})$ that $\mathcal{A}$ has not asked for. The chance of this happening is still the chance that he picked the i that $\mathcal{A}'$ picked at random in figure 6.2, i.e. $1/n$.

6.5.4 An Aggregate Signature System

The signature scheme proposed by Boneh et. al in [14] almost satisfies the requirements needed for an aggregate signature scheme. The only change needed is that the signature of a message m should be a point on the Elliptic curve, not the x-coordinate of a point.

It is based on the modified Weil pairing which is a function e that satisfies:

$$a = b \bmod q \iff e(P, bQ) = e(aP, Q)$$

when working over an elliptic curve with a point P of order q (which is the largest prime factor of the order of the curve), and where $Q \in \langle P \rangle$. The signature scheme in [14] uses a slightly modified Weil pairing by using an automorphism on the second argument to make it applicable to Decisional Diffie Hellman. The automorphism defined in [14] satisfies that:

$$\phi(aP) = a\phi(P)$$

Using this in the Weil pairing we have:

$$e(P, \phi(a\mathcal{H}(m))) = e(P, a\phi(\mathcal{H}(m))) = e(aP, \phi(\mathcal{H}(m)))$$

The idea in the signature scheme is that a is the secret key and $s = a\mathcal{H}(m)$ is a signature on message m, where $\mathcal{H}$ is the full domain hash defined in [14]. Then given the public key $R = aP$ one can check the signature by checking that:

$$e(P, \phi(s)) = e(R, \phi(\mathcal{H}(m)))$$

This requires one use of the hash function $\mathcal{H}$, and two applications of the Weil pairing. Computing the Weil pairing uses work equivalent to $\mathcal{O}(k)$ additions over the elliptic curve, where k denotes the bit size of q. The dominant computation in creating the hash of the message is a multiplication over the elliptic curve. This requires $\mathcal{O}(k)$ additions, and so the total complexity of checking a signature is $\mathcal{O}(k)$ additions.

The signature system from [14] can be extended with the extra functions needed for the aggregate signature scheme. The *KeyGen* function and the setup are identical to those found in [14]. The functions *Sign* and *Verify* are almost identical, but with the modification that *Sign* returns a point P instead of just the x coordinate of P. *Verify* checks that P satisfies the Weil pairing equation instead of checking the two candidates with x coordinate equal to the received signature.

***Setup*:** Find an elliptic curve $E/\mathbb{F}_{3^l}$, as suggested in [14, section 3.4], and find the largest prime factor q of the order of the curve. Find a point P on $E/\mathbb{F}_{3^l}$ with order q, and let $\mathcal{H}$ be a full domain hash from $\{0,1\}^*$ to $\langle P \rangle$ as defined in [14]. The global information is $(l, q, P, \mathcal{H})$.

***KeyGen*:** Given the global parameters a key pair is made by choosing a random $\alpha_i \in \mathbb{Z}_q^*$ and setting $R_i = \alpha_i P$. The public key is R_i and the private key is α_i.

***Sign*:** To sign a message m compute the hash $P_m := H(m)$ and let the signature be $s_i := \alpha_i P_m$.

***Verify*:** To verify a signature pair (s_i, m) compute $u := e(P, \phi(s_i))$ and $v := e(R_i, \phi(H(m)))$, where e is the Weil pairing and ϕ is the automorphism, both mentioned above. If $v = u$ then ACCEPT is returned, otherwise REJECT.

***ComPK*:** To compute the combination of 2 public keys pk_{X_1} and pk_{X_2} simply compute:

$$R_{X_1 \bigcup X_2} := R_{X_1} + R_{X_2} = (\alpha_{X_1} + \alpha_{X_2})P, \quad \text{where} \;\; \alpha_{X_j} = \sum_{i \in X_j} \alpha_i$$

***ComSign*:** To combine 2 signatures s_{X_1} and s_{X_2} of the same message m compute:

$$s_{X_1 \bigcup X_2} := s_{X_1} + s_{X_2} = (\alpha_{X_1} + \alpha_{X_2})P_m$$

This satisfies the associative and commutative requirements because the $+$ operator does.

***RemSign*:** To remove a signature s_i from the combined signature s_X compute:

$$s_{X \setminus \{i\}} := s_X - s_i = (\alpha_X - \alpha_i)P_m$$

It's easy to see using the properties of the modified Weil pairing described above that *ComPK* and *ComSign* actually create a corresponding public key and signature pair.

The complexity of the functions *KeyGen*, *Sign*, and *Verify* are of the order of $\mathcal{O}(k)$ additions over the elliptic curve. However, the 3 new functions *ComPK*, *ComSign*, and *RemSign* only use one addition.

6.5.5 Petitions based on Aggregate Signatures

Given a petition on the issue I, we can make the following petition system:

Setup: Given the global setup (l, q, P, H) each signer uses *KeyGen* to generate the private key α_i and the public key R_i.

Signing: For the i'th signer to sign I, it sets $P_I = H(I)$ and computes $s_{i,I} := \alpha_i P_I$. The signature $(I, s_{i,I}, \{i\})$ is sent to the petitioner.

Collecting: All the valid signatures are combined into an aggregate signature $(I, s_{X,I}, \{X\})$ using *ComSign*. When all signatures have been collected and added to $(I, s_{X,I}, \{X\})$, it is sent to the verifier.

Verifying: To test an aggregate signature $(I, s_{X,I}, X)$, the verifier creates the public key pk_X as (for simplicity $X = \{1, ..., n\}$):

$$pk_X = \mathit{ComPK}(pk_1, pk_2, ..., pk_n)$$

and tests the signature by calculating

$$\mathit{Verify}(I, s_{X,I}, pk_X)$$

If the signature is valid the number of signatures on the petition is the number of unique signers in X.

This improves the standard petition system from section 6.3 significantly because the verifier only has to combine the public keys of the signers (which is cheap compared to checking a signature) and then check one signature. The signer has the same amount of work, since the signer just needs to create a signature. The only one who has to do more work is the petitioner, who has to combine the signatures into a single master signature, but this is cheap compared to the work it has to do to verify the signatures in the first place. Given n signatures where n' of these are unique and correct we have the following complexities:

Computation:

Each signer: 1 signature.

Petitioner: Check n signatures and combine the n' signatures.

Verifier: Combine n' public keys and check 1 signature.

	Standard Signatures	Threshold Signatures	Aggregate Signatures
Work performed (additions)			
Signer	$\mathcal{O}(k)$	$\mathcal{O}(k)+\mathcal{O}(k)$	$\mathcal{O}(k)$
Petitioner	$\mathcal{O}(nk)$	$\mathcal{O}(nk)+\mathcal{O}(tk)$	$\mathcal{O}(nk)+\mathcal{O}(n')$
Verifier	$\mathcal{O}(n'k)$	$\mathcal{O}(k)$	$\mathcal{O}(k)+\mathcal{O}(n')$
Communication needed (bits)			
Signer-Petitioner	$\mathcal{O}(k)$	$\mathcal{O}(k)+\mathcal{O}(k)$	$\mathcal{O}(k)$
Petitioner-Verifier	$\mathcal{O}(n'k)$	$\mathcal{O}(k)$	$\mathcal{O}(k)+\mathcal{O}(n')$
Space needed (bits)			
Petitioner	$\mathcal{O}(n'k)$	$\mathcal{O}(tk)$	$\mathcal{O}(k)+\mathcal{O}(n')$
Attributes			
Classification	Exact	At least	Exact

Figure 6.3: Comparison between the 3 petition systems

Total communication:

Each signer to Petitioner: 1 signature.

Petitioner to Verifier: 1 signature and a list of the n' signers.

Storage space needed:

Petitioner: 1 signature and a list of the n' signers.

Classification: Exact.

The number of additions performed by the verifier is $\mathcal{O}(n'+k)$, where k is the security parameter. This is the result of n' applications of *ComPK* ($n'-1$ additions) and one application of *Verify* ($\mathcal{O}(k)$ additions). The work performed by the petitioner is $\mathcal{O}(n'k)+n'$ additions, which is $\mathcal{O}(n'k)$.

6.6 Comparison

Figure 6.3 shows a comparison of the performance in the 3 protocols. Here k denotes the bit size of the q used in the elliptic curve, n the number of signatures submitted, n' the number of unique valid signatures submitted, and t the threshold in the threshold signature system. This is based on simple implementations of both standard and threshold signatures over an elliptic curve to make it comparable to the aggregate signatures. There are a couple of notes for the table:

- The reason for the $\mathcal{O}(tk)$ work of the petitioner in the threshold signature scheme is that typically the threshold t will be large. This results in the Lagrange coefficients that will essentially be numbers modulo q. The computation of the Lagrange coefficients is not included in the complexity, since they are computed over the integers and not on the elliptic curve.

- It is assumed that the identifier of a signer is some constant size. This is used in the size of the message from the petitioner to the verifier in the standard and the aggregate signature schemes.

- The complexities in the aggregate signature scheme are based on the scenario where the petitioner adds the signatures to the aggregate signature upon receiving them. This is done after the signature has been verified as a legal signature and if it is not on the list of signatures already in the signature. The petitioner can ignore the test of whether the signature is in the list. Ignoring the test will create a slightly larger list of signers (n instead of n').

The comparison shows that the threshold signatures are the most efficient signatures with regard to the verifier. This, however, comes at the cost of only being an *At Least* petition, and the problems with adding new signers to the system. The aggregate signature system is better than the standard signatures on all accounts except petitioner work, but the extra work needed by the petitioner here is very small compared to the mandatory work needed in both cases. This means that the aggregate signature system is the best for all scenarios except those where the limitations of the threshold signature system are acceptable.

Bibliography

[1] M. Abe: *Universally Verifiable Mix-Net with Verification Work Independent of the Number of Mix-Servers*, Advances in Cryptology - EUROCRYPT '98, LNCS volume 1403, pp. 437-447. Springer Verlag, 1998.

[2] M. Abe: *Mix-networks on Permutation Networks*, Advances in Cryptology - ASIACRYPT '99, LNCS volume 1716, pp. 258-273. Springer Verlag, 1999.

[3] M. Abe, and F. Hoshino: *Remarks on Mix-network Based on Permutation Networks*, Public Key Cryptography (PKC 2001), LNCS 1992, pp. 317-324. Springer Verlag, 2001.

[4] M. Abe, and M. Ohkubo: *A Length-Invariant Hybrid Mix*, Advances in Cryptology - ASIACRYPT 2000, LNCS volume 1976, pp. 178-191. Springer Verlag, 2000.

[5] J. Algesheimer, J. Camenisch, and V. Shoup: *Efficient Computation Modulo a Shared Secret with Application to the Generation of Shared Safe-Prime Products* Advances in Cryptology - CRYPTO 2002, LNCS volume 2442, pp. 417-432. Springer Verlag, 2002.

[6] N. Asokan, H. Lipmaa, and V. Niemi: *Secure Vickrey Auctions without Threshold Trust*, Cryptology ePrint archive, record 2001/095. `http://eprint.iacr.org/`, November 2001.

[7] E. Bach, and J. Shallit: *Algorithmic Number Theory, Volume I: Efficient Algorithms*, Foundations of Computing Series. MIT Press, August 1996.

[8] J. Bar-Ilan, and D. Beaver: *Non-Cryptographic Fault-Tolerant Computing in a Constant Number of Rounds*, Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing (PODC '89), pp. 201-209. ACM Press, 1989.

[9] F. Boudot: *Efficient Proofs that a Committed Number Lies in an Interval*, Advances in Cryptology - EUROCRYPT 2000, LNCS volume 1807, pp. 431-444. Springer Verlag, 2000.

[10] O. Baudron, P.-A. Fouque, D. Pointcheval, G. Poupard, and J. Stern: *Practical Multi-Candidate Election System*, Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing (PODC '01), pp. 274-283. ACM Press, 2001.

[11] M. Bellare and S. Goldwasser: *Verifiable Partial Key Escrow*, Proceedings of the 4th ACM Conference on Computer and Communications Security, pp. 78-91. ACM Press, 1997.

[12] L. Blum, M. Blum, and M. Shub: *A simple secure unpredictable pseudo-random number generator*, SIAM Journal on Computing, volume 15, issue 2, pp. 364-383. ACM Press, May 1986.

[13] A. Boldyreva: *Efficient threshold signatures, multisignature and blind signature schemes based on the Gap-Diffie-Hellman-group signature scheme*, Cryptology ePrint archive, record 2002/118. `http://eprint.iacr.org/`, August 2002.

[14] D. Boneh, B. Lynn, and H. Shacham: *Short Signatures from the Weil Pairing*, Advances in Cryptology - ASIACRYPT 2001, LNCS volume 2248, pp. 514-532. Springer Verlag, 2001.

[15] R. Canetti: *Security and Composition of Multiparty Cryptographic Protocols*, Journal of Cryptology, volume 13, issue 1, pp. 143-202. Springer Verlag, 2000.

[16] D. Catalano, R. Gennaro, and N. Howgrave-Graham: *The bit security and Paillier's encryption scheme and its applications*, Advances in Cryptology - EUROCRYPT 2001, LNCS volume 2045, pp. 229-243. Springer Verlag, 2001.

[17] D. Chaum, and T. Pedersen: *Wallet Databases with observers*, Advances in Cryptology - CRYPTO '92, LNCS volume 740, pp. 89-105. Springer Verlag, 1992.

[18] R. Cramer, I. Damgård, S. Dziembowski, M. Hirt, and T. Rabin: *Efficient Multiparty Computations Secure against an Adaptive Adversary*, Advances in Cryptology - EUROCRYPT '99, LNCS volume 1592, pp. 311-326. Springer Verlag, 1999.

[19] R. Cramer, I. Damgård, and J. Nielsen: *Multiparty Computation from Threshold Homomorphic Encryption*, Advances in Cryptology - EUROCRYPT 2001, LNCS volume 2045, pp. 280-300. Springer Verlag, 2001.

[20] R. Cramer, I. Damgård, and B. Schoenmakers: *Proofs of Partial Knowledge and Simplified Design of Witness Hiding Protocols*, Advances in Cryptology - CRYPTO '94, LNCS volume 839, pp. 174-187. Springer Verlag, 1994.

[21] R. Cramer, M. Franklin, B. Schoenmakers, and M. Yung: *Multi-authority Secret-Ballot Elections with Linear Work*, Advances in Cryptology - EUROCRYPT '96, LNCS volume 1070, pp. 72-83. Springer Verlag, 1996.

[22] R. Cramer, R. Gennaro, and B. Schoenmakers: *A secure and optimally efficient multi-authority election scheme*, Advances in Cryptology - EUROCRYPT '97, LNCS volume 1233, pp. 103-118. Springer Verlag, 1997.

[23] R. Cramer, and V. Shoup: *Universal Hash Proofs and a Paradigm for Adaptive Chosen Ciphertext Secure Public-Key Encryption*, Advances in Cryptology - EUROCRYPT 2002, LNCS volume 2332, pp. 45-64. Springer Verlag, 2002.

[24] I. Damgård, and E. Fujisaki: *A Statistically-Hiding Integer Commitment Scheme Based on Groups with Hidden Order*, Advances in Cryptology - ASIACRYPT 2002, LNCS volume 2501, pp. 125-142. Springer Verlag, 2002.

[25] I. Damgård, and M. Jurik: *A Generalisation, a Simplification and some Applications of Paillier's Probabilistic Public-Key System*, BRICS Report Series, record 2000/45. BRICS, `http://www.brics.dk/Publications/`, December 2000.

[26] I. Damgård, and M. Jurik: *A Generalisation, a Simplification and some Applications of Paillier's Probabilistic Public-Key System*, Public Key Cryptography (PKC 2001), LNCS 1992, pp. 119-136. Springer Verlag, 2001.

[27] I. Damgård, and M. Jurik: *Client/Server Tradeoffs for Online Elections*, Public Key Cryptography (PKC 2002), LNCS 2274, pp. 125-140. Springer Verlag, 2002.

[28] I. Damgård, and M. Jurik: *A Length-Flexible Threshold Cryptosystem with Applications*, BRICS Report Series, record 2003/16. BRICS, `http://www.brics.dk/Publications/`, March 2003.

[29] I. Damgård, and M. Jurik: *A Length-Flexible Threshold Cryptosystem with Applications*, to appear in Information Security and Privacy (ACISP 2003), LNCS. Springer Verlag, 2003.

[30] I. Damgård, and M. Jurik: *Scalable Key-Escrow*, BRICS Report Series, record 2003/22. BRICS, `http://www.brics.dk/Publications/`, May 2003.

[31] I. Damgård, M. Jurik, and J.B. Nielsen: *A Generalization of Paillier's Public-Key System with Applications to Electronic Voting*, to appear in special issue on Financial Cryptography, International Journal on Information Security (IJIS). Springer Verlag.

[32] I. Damgård, and M. Koprowski: *Practical Threshold RSA Signatures Without a Trusted Dealer*, Advances in Cryptology - EUROCRYPT 2001, LNCS volume 2045, pp. 152-165. Springer Verlag, 2001.

[33] I. Damgård, and J.B. Nielsen: *An Efficient Pseudo-Random Generator with Applications to Public-Key Encryption and Constant-Round Multiparty Computation*, Manuscript.
Available from `http://www.brics.dk/~buus/papers.html`

[34] Y. Desmedt, and K. Kurosawa: *How to Break a Practical MIX and Design a New One*, Advances in Cryptology - EUROCRYPT 2000, LNCS volume 1807, pp. 557-572. Springer Verlag, 2000.

[35] T. El Gamal: *A public-key cryptosystem and a signature scheme based on discrete logarithms*, IEEE Transactions on Information Theory, volume 31, issue 4, pp. 469-472. IEEE, July 1985.

[36] A. Fiat, and A. Shamir: *How to Prove Yourself: Practical solutions to identification and signature problems*, Advances in Cryptology - CRYPTO '86, LNCS volume 263, pp. 186-194. Springer Verlag, 1987.

[37] P.-A. Fouque, G. Poupard, and J. Stern: *Sharing Decryption in the Context of Voting or Lotteries*, Financial Cryptography (2000), LNCS volume 1962, pp. 90-104. Springer Verlag, 2001.

[38] Y. Frankel, P. MacKenzie, and M. Yung: *Robust Efficient Distributed RSA-key Generation*, Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing (STOC 98), pp. 663-672. ACM Press, 1998.

[39] A. Fujioka, T. Okamoto, and K. Ohta: *A Practical Secret Voting Scheme for Large Scale Elections*, Advances in Cryptology - AUSCRYPT '92, LNCS volume 718, pp. 244-251. Springer Verlag, 1993.

[40] E. Fujisaki, and T. Okamoto: *Statistical Zero Knowledge Protocols to Prove Modular Polynomial Relations*, Advances in Cryptology - CRYPTO '97, LNCS volume 1294, pp. 16-30. Springer Verlag, 1997.

[41] O. Goldreich, S. Micali, and A. Wigderson: *How to play ANY mental game or A Completeness Theorem for Protocols with Honest Majority* Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing (STOC 87), pp. 218-229. ACM Press, 1987.

[42] O. Goldreich and V. Rosen: *On the security of modular exponentiation with application to the construction of pseudorandom generators*, Cryptology ePrint archive, record 2000/064. `http://eprint.iacr.org/`, December 2000.

[43] J. Groth: *Extracting Witnesses From Proofs of Knowledge in the Random Oracle Model*, BRICS Report Series, record 2001/52. BRICS, `http://www.brics.dk/Publications/`, December 2001.

[44] L. Guillou, and J.-J. Quisquater: *A Practical Zero-Knowledge Protocol fitted to Security Microprocessor Minimizing both Transmission and Memory*, Advances in Cryptology - EUROCRYPT '88, LNCS volume 330, pp. 123-128. Springer Verlag, 1988.

[45] M. Hirt, and K. Sako: *Efficient Receipt-Free Voting based on Homomorphic Encryption*, Advances in Cryptology - EUROCRYPT 2000, LNCS volume 1807, pp. 539-556. Springer Verlag, 2000.

[46] M. Jakobsson, and A. Juels, *An optimally robust hybrid mix network*, Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing (PODC '01), pp. 284-292. ACM Press, 2001.

[47] A. Kiayias, and M. Yung: *Self-Tallying Elections and Perfect Ballot Secrecy*, Public Key Cryptography (PKC 2002), LNCS 2274, pp. 141-158. Springer Verlag, 2002.

[48] W. Mao: *Publicly Verifiable Partial Key Escrow*, Information and Communications Security (ICICS'97), LNCS 1334, pp. 409-413. Springer Verlag, 1997.

[49] S. Micali: *Guaranteed Partial Key Escrow*, MIT laboratory of computer science, Technical Memo 537. MIT Press, September 1995.

[50] S. Micali and A. Shamir: *Partial Key Escrow*, Manuscript. February 1996.

[51] T. Okamoto, and S. Uchiyama: *A New Public-Key Cryptosystem as Secure as Factoring*, Advances in Cryptology - EUROCRYPT '98, LNCS volume 1403, pp. 308-318. Springer Verlag, 1998.

[52] P. Paillier: *Public-Key Cryptosystems based on Composite Degree Residue Classes*, Advances in Cryptology - EUROCRYPT '99, LNCS volume 1592, pp. 223-238. Springer Verlag, 1999.

[53] B. Schoenmakers: *A Simple Publicly Verifiable Secret Sharing Scheme and Its Application to Electronic Voting*, Advances in Cryptology - CRYPTO '99, LNCS volume 1666, pp. 148-164. Springer Verlag, 1999.

[54] A. Shamir: *How to Share a Secret*, Communications of the ACM, volume 22, issue 11, pp. 612-613. ACM Press, November 1979.

[55] A. Shamir: *Partial Key Escrow: A New Approach to Software Key Escrow*, Key Escrow Workshop. National Institute of Standards and Technology, September 15, 1995.

[56] J. Shaoquan and Z. Yufeng: *Partial Key Escrow Monitoring Scheme*, Cryptology ePrint archive, record 2002/039. `http://eprint.iacr.org/`, March 2002.

[57] V. Shoup: *Practical Threshold Signatures*, Advances in Cryptology - EUROCRYPT 2000, LNCS volume 1807, pp. 207-220. Springer Verlag, 2000.

[58] A. Waksman: *A Permutation Network*, Journal of the ACM, volume 15, issue 1, pp. 159-163. ACM Press, January 1968.

Recent BRICS Dissertation Series Publications

DS-03-9 **Mads J. Jurik.** ***Extensions to the Paillier Cryptosystem with Applications to Cryptological Protocols*****. August 2003. PhD thesis. xii+117 pp.**

DS-03-8 **Jesper Buus Nielsen.** ***On Protocol Security in the Cryptographic Model*****. August 2003. PhD thesis. xiv+341 pp.**

DS-03-7 **Mario José Cáccamo.** ***A Formal Calculus for Categories*****. June 2003. PhD thesis. xiv+151.**

DS-03-6 **Rasmus K. Ursem.** ***Models for Evolutionary Algorithms and Their Applications in System Identification and Control Optimization*****. June 2003. PhD thesis. xiv+183 pp.**

DS-03-5 **Giuseppe Milicia.** ***Applying Formal Methods to Programming Language Design and Implementation*****. June 2003. PhD thesis. xvi+211.**

DS-03-4 **Federico Crazzolara.** ***Language, Semantics, and Methods for Security Protocols*****. May 2003. PhD thesis. xii+160.**

DS-03-3 **Jiří Srba.** ***Decidability and Complexity Issues for Infinite-State Processes*****. 2003. PhD thesis. xii+171 pp.**

DS-03-2 **Frank D. Valencia.** ***Temporal Concurrent Constraint Programming*****. February 2003. PhD thesis. xvii+174.**

DS-03-1 **Claus Brabrand.** ***Domain Specific Languages for Interactive Web Services*****. January 2003. PhD thesis. xiv+214 pp.**

DS-02-5 **Rasmus Pagh.** ***Hashing, Randomness and Dictionaries*****. October 2002. PhD thesis. x+167 pp.**

DS-02-4 **Anders Møller.** ***Program Verification with Monadic Second-Order Logic & Languages for Web Service Development*****. September 2002. PhD thesis. xvi+337 pp.**

DS-02-3 **Riko Jacob.** ***Dynamic Planar Convex hull*****. May 2002. PhD thesis. xiv+110 pp.**

DS-02-2 **Stefan Dantchev.** ***On Resolution Complexity of Matching Principles*****. May 2002. PhD thesis. xii+70 pp.**

DS-02-1 **M. Oliver Möller.** ***Structure and Hierarchy in Real-Time Systems*****. April 2002. PhD thesis. xvi+228 pp.**

DS-01-10 **Mikkel T. Jensen.** ***Robust and Flexible Scheduling with Evolutionary Computation*****. November 2001. PhD thesis. xii+299 pp.**