Aalto University
School of Science
NordSecMob Master's Program

Sergio Andrés Figueroa Santos

A Cost-Effective Approach to Key Management in Online Voting Scenarios

Master's Thesis
Espoo, 1st June 2016

Supervisors:	Helger Lipmaa, Ph.D., University of Tartu
	Tuomas Aura, Ph.D., Aalto University
Instructor:	Sven Heiberg, M.Sc., Cybernetica AS

Aalto University School of Science Degree Programme in Computer Science and Engineering NordSecMob Master's Program	ABSTRACT OF THE MASTER'S THESIS
Author: Sergio Andrés Figueroa Santos	
Title: A Cost-Effective Approach to Key Management in Online Voting Scenarios	

Number of pages: 45	Date: 1st June 2016	Language: English
Professorship: Security and Mobile Computing (T3011)	Code: T-110	
Supervisor: Helger Lipmaa, Ph.D., University of Tartu		
Supervisor: Tuomas Aura, Ph.D., Aalto University		
Advisor: Sven Heiberg, M.Sc., Cybernetica AS		

Abstract: The problem of key management is an information security issue at the core of any cryptographic protocol where identity is involved (e.g. encryption, digital signature). In particular for the case of online voting, it is critical to ensure that no single actor (or small group of colluding actors) can impact the result of the election nor break the secrecy of the ballot.

The concept of threshold encryption is present at the core of many Multi-Party Computation (MPC) protocols, even more so in the scenario of online voting protocols. On the other hand, the generic key management problem has led to the design of certifiably secure hardware for cryptographic purposes. There are three families of these kind of *designed for security* devices: Hardware Security Modules (HSMs), Trusted Platform Modules (TPMs) and smart cards.

Since smart cards both offer reasonable prices and expose an API for development, this document evaluates different approaches to implement threshold encryption over smart cards to support an electoral process.

Keywords: online voting, key management, threshold cryptography, smart cards, untrusted dealer

Contents

Acknowledgements

I would like to extend my appreciation words to Mr. Sven Heiberg and Prof. Helger Lipmaa for stepping out of their comfort zone in my pursue for a project that would have both academical and practical value. I want to thank Arnis Paršovs and Ivo Kubjas for their selfless and timely advise. Their knowledge of academic research and Java Card led the way for my efforts.

I must put also a word of recognition on behalf of a group of online users. We did not know each other beforehand and I had nothing to offer them. And, however, they were always willing to engage in lengthy conversations about topics that are otherwise poorly documented, seldom updated or would have in any other manner a lengthy learning curve. Thanks to Maarten Bodewes, Geoffroy Couteau, *thotheolh* and so many other helpful and skilled contributors to the StackExchange network. Your feedback helped me to find the right way in this maze.

And finally, to Kerstin. She always had the right word to fuel my mood, my thought process and my English grammar. She listened, inquired and suggested. She gave a second opinion that enriched my whole approach, and showed me the way whenever I felt stuck. Enthusiastic and clever, she gave a soul to this work. *Thanks.*

1 Introduction

This thesis describes the design process of a solution to enable key storage that can be used to hold legally binding electoral processes.

The use of information technologies to support elections enables voting processes to provide proofs of transparency to voters that are more reliable than any traditional system, at the expense of clarity: the level of previous knowledge required to understand the nature and reliability of a cryptographic model is higher and more specific than it would be for paper ballot, to name an example.

As it is to be expected, the implementation of cryptographic protocols for the specific scenario of electronic voting inherits all the concerns that need to be addressed for any cryptographic implementation: e.g. the selection of suitable parameters and the specific algorithms, the decision of using well-proven libraries against implementing the algorithms from scratch, identifying a suitable randomness generator and extracting randomness from it adequately, or protecting the secret key used in the algorithm.

The latter will be the major problem addressed during this work. The present document will explore the problem of key storage in general, and its particularities within the online voting scenario. It analyzes the viability of introducing threshold encryption schemes using smart cards as a platform for security critical operations. As a mechanism to validate the approach proposed, this document evaluates its suitability within the Estonian Internet Voting scheme, which has been used for legally binding elections throughout more than a decade.

This document presents an overview of the general problem of key management on chapter 2. Chapter 3 presents the specific key management requirements within an online voting scenario, using the experience of the Estonian Internet Voting scheme as a use case. A qualitative framework for comparing key management strategies and a viability analysis of alternatives comprise the chapter 4. Finally, chapter 5 summarizes the results obtained and introduces a wide range of questions that can lead future research in related areas.

Although this document is directed to an audience familiar with cryptography and/or voting systems, it does not assume any prior knowledge in terms of specific notation or concepts. In that regard, the annexes provide a guide about specific terminology used in the document. Annex A lists mathematical symbols and variable names, annex B expands all the acronyms of technologies, techniques and institutions, whereas annex C contains a short introduction to essential cryptographic primitives.

2 Overview

The definition of a cryptographic primitive is typically represented as a network protocol, where the communication channel is hostile and the parties, conventionally labeled as Alice and Bob, are trustworthy. In some cases, the idea that the other actors can deliberately attempt to deviate from the protocol is acknowledged and modeled.

For simplicity in some cases, and for necessity in others, these models rely upon certain assumptions. Ignoring their existence leads to some of the least evident and most harmful bugs and vulnerabilities in cryptographic implementations.

There are two major assumptions that support the majority of the functionality in cryptographic primitives:

1. It is possible to choose an element from an arbitrary set at random with uniform probability (true randomness).

2. Any actor in a cryptographic protocol is a deterministic black box that can keep a secret and only discloses the information determined by the protocol.

In particular, the second assumption implies veiled requirements that cannot be met by any implementation. What is Alice? Is it the user making use of the system, or is it the computer, or the program running on top? How can "Alice" be trusted to follow the protocol? What happens when several copies of secret data (key, password, etc.) are made? *How can a secret key be kept secret?*

As a key is a kind of secret that is critical for several cryptographic primitives, these questions lead to the concept of *key management*. It consists of following the lifecycle of a key in order to enable its availability while protecting its secrecy and integrity. The objective of this section is to describe the importance of key management, the threats that it must overcome and the existing approaches to the problem.

2.1 Importance of Cryptographic Keys

A cryptographic scheme in the malicious model can offer meaningful guarantees of security to Alice *if Alice is honest*. The honesty of Alice is typically described as her adherence to the protocol. However, modern cryptographic protocols with non-trivial security parameters cannot be executed correctly and efficiently by a human mind. Alice does not execute the protocol directly, but through a proxy, an information system. In that scenario, the "moral" requirement of honesty becomes a technical one.

If Alice is *technically honest*, she will follow the protocol rigorously up to every constraint: correct ordering of the messages will be observed, the variables will be chosen within the given range, the calculations will be performed accurately and every nondeterministic action will be as nondeterministic as possible. From the functional point of view, correctness is enough.

However, from the information security point of view, correctness is not sufficient. Since the main objective of cryptographic protocols is to enforce specific restrictions (e.g. "only Alice can read the content", "the tally can be generated only by the electoral committee"), it is also important for any statement to ensure that *no information is leaked* beyond the intended functionality.

In particular, the identity of Alice is usually represented by a secret piece of data: a *key*. In cryptographic protocols, the security of the key can determine the security boundaries of the system. However, said importance is seldom accounted for within the design of the protocol. In the context of most cryptographic definitions, the key is generated according to certain constraints (e.g. a random element in $\mathbb{Z}_p$) and then stored in a "secure void", where it can be accessed when it is required by the protocol.

There is, nevertheless, no secure void within an information system. A key is a string of bits that needs to be stored and retrieved to working memory and needs to be backed up to withstand physical failure. A key is a physical entity that can exist in different places at the same time. It is also intended to be the only obstacle for an adversary attempting an attack within an otherwise secure protocol. In short: a cryptographic protocol implementation is only as strong as the protection around its key material.

The security of a strong cryptographic protocol is bounded by the weakest path to *any* copy of the secret key. The repertoire of attack vectors that aim to recover the secret key within an encryption system is rich and diverse.

2.2 Attacks Vectors for Key Retrieval

There is active research about how to break keys whose security relies on mathematical assumptions. For instance, since a private RSA key can be obtained by factoring the public key, improving factoring can be a way to break the system. However, there are no efficient algorithms for factoring so far, and therefore RSA is still secure as a scheme (given the right choice of parameters).

On the other hand, the implementations of cryptographic functions are seldom as careful and isolated as their mathematical models. Bugs, malpractice and unexpected variables reduce the difficulty of guessing a key. This section describes different attack vectors that leverage this fact.

2.2.1 No Randomness

The difference between encoding and encryption may be subtle, but essential. An encoded message can be decoded by anyone who knows (or can derive) the rules. There is no secret *key*. An encrypted message, on the other hand, can only be decrypted (theoretically) by the holder of the key. A similar logic applies for other keyed primitives, such as MAC integrity checks.

It is a common anti-pattern to hard-code a default constant value within the code of an application in order to use it as a secret key. If the key used within an encryption scheme is not random and cannot be changed easily, the encryption scheme is being used for encoding. If a software product uses it as a key, its retrieval is, at best, as hard as reverse engineering the application.

2.2.2 Poor Randomness

Any keyed algorithm relies at least once on the random generation of a value k extracted from the universe of values K with nearly uniform probability. In other words, that $\left(\forall k^* \in K : Pr[k = k^*] \approx \frac{1}{|K|}\right)$.

The definition of what is random is elusive, although there are tests to assess the randomness of a string. For this reason, the difference between randomness that is good enough for a simulation or the decisions in a game and randomness suitable for cryptographic use is often subtle and not properly analyzed. The use of poor random generators for cryptographic purposes has been repeatedly documented.

From the deliberate construction of backdoors [1] to API misuse [2], the usage of random number generators that are not *random enough* is a major problem in present day cryptographic implementations [3].

Furthermore, without access to a true randomness source, a good randomness function, i.e. a Cryptographically Secure Pseudo Random Number Generator (CSPRNG) a *seed* is required: a value that initializes the CSPRNG. The CSPRNG is, after initialization, deterministic: given a seed, the string of random values produced will always be the same. Examples of bad seeds are a constant integer or a function of the system time, and yet the most remarkable mistake regarding seeds is repetition: to use the same seed in two instances of the CSPRNG. One scenario is the non-random presetting of a key: choosing an arbitrary but fixed seed defeats its purpose. Another, more subtle alternative appears when the seed is derived from network, physical and software parameters, since these parameters do not have enough variance/randomness to act as an acceptable seed.

In both cases, the choice of a weak seed will lead to different independent systems inadvertently sharing keys and the possibility for attackers to calculate it themselves.

2.2.3 Access Control

Access control is, after cryptography, the most efficient family of controls to ensure confidentiality and integrity requirements. It is possible to define a flexible set of actors, actions, assets and permissions to model fine grained policies.

Unlike cryptography, access control is not applied directly on the protected asset, but on its containers: e.g. operating systems, physical facilities, application servers or network infrastructure. As a result, if access control is overridden, the protected asset is exposed.

Even under the assumption of a flawless implementation of access control mechanisms, its configuration is neither static nor trivial. As a result, its maintenance becomes a highly demanding operational task that is likely to be neglected when it interrupts the execution of more productive ones. In other words: business continuity is likely to antagonize access control, deteriorating over time the accuracy with which access control rules reflect the business rules, diminishing its effectiveness as a control.

Another security limitation of access control schemes is that they can fail silently. For example, a firewall that is not filtering packages will work as well as one that only accepts authorized requests *as long as only authorized requests are sent.*

As a result, relying on access control mechanisms to protect a private key can be considered a good practice, albeit insufficient. The key can be leaked inadvertently and the policies might even allow access to another existent but unauthorized user (i.e. escalation of privileges).

2.2.4 Application Vulnerabilities

Even when a cryptographic algorithm is secure according to a mathematical model, its implementation requires interaction with existing technologies and their limitations: response times, APIs, data types, abstractions, frameworks and legacy applications. As a result, the vulnerabilities of the underlying technology are inherited by the implementation and need to be addressed or accepted. For example, a cryptographic library in C needs to acknowledge the possibility of buffer overflow attacks and prevent information leakage by exploitation of the same.

One of the biggest recent examples of the extent to which application dependencies can lead to major security incidents is the vulnerability CVE-2014-0160, publicly known as Heartbleed. The vulnerability affected OpenSSL, a major open source library for SSL support. Under certain conditions, the bug enabled the remote recovery of server side information, including in some scenarios the secret key of the server. The number of vulnerable Internet servers varies according to the measurement criteria, but the vulnerability existed for years before being revealed and affected a vast amount of the most visited websites on the Internet [4].

Heartbleed is not the only internet-wide major vulnerability disclosed in the last years, and the fact that it remained undiscovered for so long reveals how subtle and critical the security assessment of an application can be.

2.2.5 Poor Protection of Multiple Copies

It is not realistic to assume that there is one single instance of a cryptographic key that can be used arbitrarily and safely without moving it or copying it. In a real computer, the key can be stored in secondary storage. When it is needed, a copy is generated in main memory. Furthermore, the requirements of availability may lead to store backups of the key in different information systems (either cold or hot backup). Finally, for practical or economical reasons, the same key can be used for different solutions that implement a variety of protocols.

Since each copy of the key is identical, each of them has exactly the same value. For instance, a digital document can be signed with any of the copies of the key with exactly the same validity. The main impact of this remark is the fact that the key is only as secure as the most vulnerable of its containers.

If the key is stored in an encrypted hard drive, but it is not deleted from RAM once it is used, it may be easier for an attacker to dump the RAM than to decrypt the content of the disk. Likewise, if the cold backup is stored in a warehouse that is only secured with a lock, it will be easier for an attacker to steal that copy and leave no digital trace of the theft.

2.2.6 Malpractice

One of the least obvious consequences of implementing a cryptographic protocol in a computer is that other processes, unrelated to the protocol itself, can jeopardize the key to the point of making the whole solution superfluous.

The security malpractices that can leak the key are diverse in nature, propensity and exploitability. Failure to erase the key from a hard drive that is being disposed of enables a dumpster diver to access the key.

Additional examples of poor security practices include negligent access control configuration (e.g. `chmod 777` in Unix systems, used as a way to debug errors related to system permissions), unsafe transmission of the key material or inadvertent publication in a public repository.

2.2.7 Side Channel Attacks

The field of side channel attacks is the result of the realization that every cryptographic primitive is executed within a physical machine that may be used for more than a single purpose. As a result, it assesses the implications of that observation in terms of the security of that primitive. The possibility of retrieving a cryptographic key from the analysis of the temperature, power consumption or environmental noise defies the validity of the abstraction in which computer scientists work. When security is involved, a black box that follows a protocol deliberately omits details that can falsify any security assertion.

Furthermore, the analysis of the effects of concurrency, processing scheduling, latency and shared memory, among other characteristics of modern computational environments, provides additional information to adversaries that cannot be fully predicted by mathematical models.

The possibility of attacking the underlying system that implements a cryptographic primitive to break its security is a latent and inherently chaotic threat that needs to be addressed as part of a cryptographic implementation.

2.3 Key Storage Solutions

The variety and subtlety of attack vectors on a key do not imply that the key is inexorably exposed and that all keyed cryptographic primitives are insecure. It does imply, however, that key management is a critical issue and the decisions made around it may impact the security claims made about the whole system.

This section explores solutions to the problem of key storage and the guarantees that can be offered by each of them.

2.3.1 Cleartext

A cryptographic key can be represented in a file. One of the most commonly used representations for that is the ASN.1 notation. ASN.1 is typically used to encode the attributes of a cryptographic key in a multiplatform, standardized way. A file that is encoded correctly can be copied, transmitted and read like any other file within the file system.

This representation is, however, insufficient for the storage of secret data. A cleartext key can be leaked or tampered with trivially, and the detection of any compromise is unlikely, even in the long term. Although the implementation of file system access control measures can reduce the attack surface, this representation is still too vulnerable against experienced attackers.

2.3.2 PKCS#12

The Public Key Cryptography Standards (PKCS) define the parameters for encoding cryptographic material used for public key schemes, including certificates, revocation lists, certificate signing requests and private keys. In particular, the standard PKCS#12 defines the concept of a *keystore*. The standard is also described in RFC 7292 [5]. A single keystore may contain different *entries*, identified by a string knows as *alias*.

One of the features that differentiate a keystore as the container of sensitive material from a normal binary file is the definition of "several privacy and integrity modes" [5]. As a result, the transmission and storage of personal identity information (i.e. cryptographic keys and associated information) can be protected with a password derived key or another key pair.

Although it is theoretically the most powerful option, the use of another key pair just transfers the problem, since the new key will also need protection. As a result, the popular implementations of the protocol (in particular, it is the default mechanism for the retrieval of a private key from the major Internet browsers and operating systems) favor the use of password protected keystores.

As a general idea, a password is used in the PKCS#12 standard to derive a symmetric encryption key. However, the standard offers different levels of granularity, which may or may not use the same password. The first separation occurs between privacy and integrity: the encryption and MAC algorithms might use the same password, or require two different. Furthermore, in addition to global level passwords, it is possible to establish passwords specific for each entry in the keystore.

Although the use of key derivation functions reduces the viability of password guessing attacks, the use of passwords enables usability malpractices that can lead to a leak of the private key. Some of the pitfalls, enunciated by [6] more than a decade ago and still present in the industry, lead to the conclusion that no security critical decision should be left to the discretion of the final user, if it can be avoided. Some examples were the reuse of the same key in different environments or sending the password along with the keystore.

Another specific scenario involves access to the key material by an automated process. Be it a server that supports SSL communication, or an application compatible with PGP, the problem appears when asking the user every time for the password is not feasible or makes no sense for the specific process. The storage of the password becomes necessary but problematic. As a result, the password is stored in locations that are not designed to store sensitive information, such as a shell script that calls the encryption process, or an unprotected text file. In general, a password has the same limitations and privacy requirements as a cryptographic key, with the additional issue of being human readable, and could therefore be leaked in unexpected ways. Section 6 of RFC 7292 states just that limitation and suggests guidelines to maximize the effectiveness of password based protection.

It must also be noted that the PKCS family focuses on containers and definitions for public key encryption material, and thus could be unsuitable for the storage of other kinds of cryptographic material.

2.3.3 Trusted Platform Modules (TPM)

A Trusted Platform Module (TPM) is a purpose-specific processor designed to support cryptographic operations in general purpose computers, standardized by ISO-11889 [7]. Along with security controls against unauthorized use and cryptographic support, its architecture supports random number generation and key storage.

The TPM is designed to isolate the storage and usage of cryptographic material, as well as the randomness generation, from the data and instructions, reducing the possibility of key leakage due to API misuse. Given its increasing presence in consumer products, it is used to support cryptographic functionality such as BitLocker, the application for Full Disk Encryption available in recent Windows versions.

The most recent version of TPM, 2.0, enables support for the most common cryptographic algorithms, such as RSA, SHA-1, SHA-256 and elliptic curve. However, no specific algorithm is mandatory. In terms of physical portability, a TPM is as portable as the computing system that contains it.

2.3.4 Smart Cards

Smart cards are very limited computing devices that rely on a card reader as a power and communication source. Although they are not designed to enable intensive operations (neither in terms of time nor space), they often include built-in support of standard cryptographic primitives. As a result, they are a popular, portable and isolated environment for the storage and usage of cryptographic keys.

The standard ISO/IEC 7816 [8] defines the features of contact, contactless and hybrid smart cards, and their cryptographic capabilities are often certified to some level of the FIPS 140-2 standard [9]. As a consequence, smart cards are used as a strong authentication mechanism. On the other hand, their functionality is neither scalable nor very flexible, and their portability enables non-conventional attack vectors, mainly within the realm of side channel attacks.

A popular development platform for smart cards is Java Card, which is designed as a subset of the Java language for resource constrained devices. Although it shares its syntax with Java, Java Card suffers from a limited API and extremely limited storage and processing power. It is designed to support classical cryptographic algorithms, such as AES, RSA and the SHA hash family. However, it does not offer support for homomorphic encryption schemes, including ElGamal. An application developed for Java Card is known as an *applet*.

2.3.5 Hardware Security Modules (HSM)

A stronger instance of the separation between cryptographic and business operations is known as a Hardware Security Module (HSM). An HSM is an appliance that implements tamper-evident and tamper-resistant mechanisms around an application-specific cryptographic computer.

HSMs are considered one of the most efficient mechanisms for the secure storage and usage of cryptographic material, supporting the operation of critical processes such as online banking, military communications or the operation of a certification authority. They are subject to strict security certifications, of which FIPS 140-2 [9] is the most prevalent. The modules include mechanisms that enable high availability, secure key backup and efficient encryption.

Despite being recognized in the industry as the most powerful option, HSMs are not cost efficient for every critical process, even when the information being protected is highly sensitive. Furthermore, very few HSMs enable customized functionality, and the ones that do are even more expensive. As a result, the search for cheaper ways to obtain similar levels of assurance and greater levels of customization becomes an important research field.

2.4 Secret Sharing and Threshold Encryption

The approach for key protection described up to this point is to treat a key as a single point of failure. If there are several copies of the key, each of them is critical, since the leak of any of them compromises the security of the system as a whole, regardless of the controls implemented around the other copies. In this section, the aim is to distribute the risk so that, if the controls around one copy are compromised, the security of the system remains resilient.

The most popular approach in this direction is Shamir's Secret Sharing Scheme [10], remarkable as a cryptographic scheme that is both practical and information-theoretically secure. In this approach, the key is split into shares that can be used to reconstruct the key later on.

This approach introduces two concepts: *secret sharing* and *threshold*.

A secret sharing scheme involves two phases: the *generation* of the shares and the *reconstruction* phase. For the generation of the shares, two parameters are defined: the number of shares n and the threshold t. The shares are generated from a secret s and distributed among different trustees. The reconstruction phase allows the calculation of s if and only if at least t shares are available. *After the reconstruction phase, the secret is no longer shared.*

The selection of the values t and n depends on each specific scenario, strongly determined by its practical constraints. How many trustees will receive a share? Should one trustee receive more shares than another? Which is the minimum amount of trustees required to perform the threshold reconstruction? It is essential to note that different choices for these values will result in different implications.

A high threshold will reduce liability in the case where few shares are compromised or a group of the trustees collude. If more shares are required (i.e. if t is bigger), a collusion or an attack will need to compromise a wider attack surface. On the other hand, if the threshold is too close to the total number of shares, the integrity and availability of the key is jeopardized. In the extreme case, if $n = t$, it would suffice to lose or corrupt one share to make the system unusable.

Threshold schemes, thus, offer resilience *or* security, but for most reasonable values, these features constitute a trade-off.

2.4.1 Shamir's Secret Sharing

As mentioned earlier, Shamir's Secret Sharing scheme is one of the first and most popular secret sharing schemes [10]. Given a secret value $s \in \mathbb{Z}_p$, it produces a set $S = \{S_1, ..., S_n\}$ as follows:

Algorithm 1: `Shamir.SecretSharing`

```
input  : $s, t, n$
output: $a, S$
1  $a_0 \leftarrow s$;
2  for $i \leftarrow 1$ to $t-1$ do
3      $a_i \xleftarrow{\$} \mathbb{Z}_p$ ; // Create [t] random coefficients
4  for $i \leftarrow 1$ to $n$ do
5      $x_i \leftarrow getX(i)$ ; // Creates a value of [x] for the [i]-th share
       // Polynomial evaluation:
6      $y_i \leftarrow a_0$;
7      for $j \leftarrow 1$ to $t-1$ do
8          $y_i \leftarrow s_i + a_j \cdot x_i^j$
9      $s_i \leftarrow (x_i, y_i)$
10 $a \leftarrow a_0, ..., a_{t-1}$;
11 $S \leftarrow s_1, ..., s_n$
```

The only requirement for the function $getX(i)$ is to produce different values in $\mathbb{Z}_p$ for different inputs of i. In particular, defining $getX(i) = i$ is acceptable. As a result, the indexes can be public.

A set $S' = \{S'_1, ..., S'_t\}$ such that $S' \subseteq S$ can be used to reconstruct the secret s using Lagrange interpolation as follows:

$$\texttt{Shamir.SecretReconstruction}(S' = (X', Y')) = s = L(0) = \sum_{j=0}^{k} Y'_j \prod_{m=0, m \neq j}^{k} \frac{X'_m}{X'_m - X'_j} \tag{1}$$

2.4.2 Verifiable Secret Sharing

In the basic Shamir's secret sharing scheme, the trust in the dealer is absolute: the trusted parties receive values in $\mathbb{Z}_p$ that are indiscernible from random. The definition of the scheme ensures that a group of at least k parties will be able to produce a secret $s*$, but there is no way to prove to the shareholders that $s = s*$.

This is particularly critical when the secret is to be used as the private key of an encryption scheme. If the dealer creates a key pair (sk, pk), publishes pk and then distributes n random values in $\mathbb{Z}_p$ instead of correctly calculated shares, all the messages encrypted using pk will be lost, since (with overwhelming probability) those shares will not produce the value sk. In the case of a voting process, the error could only be detected after the election is over and all the votes would be lost.

The approach to reduce the impact of this observation is known as *Verifiable Secret Sharing* (VSS)[11]. The aim of VSS is to ensure consistency by verifying that the shares are not randomly chosen values. In the case of the key pair, the objective is to prove that an arbitrary share s_i is a valid share for the secret key sk. In particular, this validation should be possible without requiring access to the other shares, disclosure the value sk nor initiation of the reconstruction process.

The models for VSS vary depending upon the nature of the secret. For instance, in the case of a key pair (sk, pk), the relationship between the secret sk and the public value pk can be used to prove properties of sk without revealing its value. The scope of VSS is defined in [12] in two points:

1. *Prevent a malicious dealer* from distributing *incorrect* shares to the participants (i.e. send $s* \notin S$, where $S =$`SecretSharing`(s) for an arbitrary secret s).

2. *Prevent a malicious trusted party* from contributing with an incorrect share to `SecretReconstruction` (i.e. for a user i, to send $s* \neq s_i$).

2.4.3 Distributed Key Generation

VSS is an important building block in threshold cryptography, but its functional purpose is unclear until it is looked at in the context of Distributed Key Generation (DKG). The goal of DKG is to securely distribute shares of a secret key among authorized users. The exact extent of the term *securely* depends on the different scenarios considered. Specifically, the security model depends on the existence or absence of a *trusted dealer*.

If there is a trusted dealer, its function is to create the secret value and distribute its shares using a VSS scheme. If there is no trusted dealer, each of the trusted parties contributes to the calculation of a shared secret. An example of a DKG protocol without a trusted dealer is discussed in section 4.4.2.

2.5 Cryptography and E-Voting

This section shifts the perspective from key management to the other end of the problem of this thesis: online voting. The description of voting as an information security problem shares so many elements with the description of a Multi-Party Computation (MPC) protocol that both research areas are intricately entangled and the results of one often are a generalization (or an instance) of the other.

A voting process involves a simple objective, the aggregation of the points of view of a group of valid voters into a measurable result (the tally). Given a set of previously defined rules, the functional side of the solution is not likely to pose a hard engineering challenge. The security of the process, however, is complicated by a diverse set of actors who, guided by a wide range of motivations, may collude to influence the value of the tally.

Each of the entities involved in a voting process has the right or the duty of fulfilling a particular role in the process. For example, the voters may cast a vote, the registration authority must issue identities and the tallying authority has to aggregate the values of the votes cast. However, each of the parties might be willing to exceed their influence in the process, and therefore each of them has reason to mistrust the others.

When the dynamics of the process lead to requirements that seem to be contradictory (such as ballot secrecy and verifiability), the attempts to satisfy them call for the use of specialized cryptographic approaches. The techniques, thoroughly documented in [13], include homomorphic encryption, mix-nets, blind signatures, Zero Knowledge Proofs and commitments.

2.5.1 Availability of Technology

It is important to remark that while there is broad support of cryptographic functionality, both in the form of APIs and end user products, said support is focused on the protection of the world driven by content and multimedia that represents the majority of all Internet traffic.

A survey on encryption products was published in February 2016 by a team led by the researcher Bruce Schneier [14]. Although the researchers themselves acknowledge that the content might be incomplete or inaccurate (the research relied on crowdsourcing as one of the sources for the data), the survey aims to reflect the main trends in terms of cryptography implementations. The survey identifies 865 solutions that range from incipient projects to mature corporate solutions, enabling encryption in different categories, including file encryption, secure file transfer, email and IM encryption, cryptographic libraries, secure calling and decentralized Internet tools (anonymizers, cryptocurrencies, onion browsing, proxies, etc.). None of them involves explicitly Multi-Party Computation or Zero Knowledge Proofs [1]. There are two open source C++ projects that claim to support homomorphic operations. One of them is labeled by its authors as "mostly meant for researchers"[15].

In particular, there are no broadly accepted and tested implementations. While modern programming languages have native API to support cryptographic operations (e.g. Java Cryptography Architecture) and libraries like BouncyCastle [16] and OpenSSL [17] are widely used, tested and maintained, implementations related to MPC functionality

[1]Three of the products use the term *zero knowledge* while describing what is commonly known as *end-to-end encryption*. There is no reference to Zero Knowledge Proofs associated with them.

are scarce and mostly proofs of concept, which means that they are not designed to be used securely in production environments.

There are implementations of MPC across the world, but the fact that they are not identified under the spotlight of such a broad survey suggests that the knowledge is very centralized and inaccessible.

3 Key Management in the Estonian Internet Voting Process

The objective of this document is to present a key management scheme suitable for electronic voting. As a study case, it examines the case of Estonian Internet Voting, which has been used as a valid channel for casting legally binding votes since 2005 [18].

3.1 Internet Voting in Estonia

Since its introduction in 2005 the adoption of online voting in Estonia has been slow, but is steadily increasing. During the European Parliament Elections of 2014 and the National Parliamentary Elections of 2015, more than 30% of the votes were cast via this channel.

The main characteristic of the Estonian model is that elections are held over the Internet, as opposed to an event held within controlled venues[2]. That design choice implies the need of a carefully defined security model, comprising both strong mathematical tools and security procedures. In fact, given that the reliability of the model depends on the right selection and interaction of complex tools, as opposed to simple concepts such as ballot recounting, there have been evaluators who are not satisfied with the Estonian model [19].

One of the measures that can be taken to address the criticisms is to ensure that the technical implementation is as faithful to the proven security models as possible. Within this approach, the protection of the secret key material becomes a critical aspect.

3.1.1 Current Status of Internet Voting

The Estonian identification card gives every citizen two certified key pairs for authentication and document signature. This mechanism enables the creation of a secure channel with bidirectional strong authentication between organizations and individuals, in order to enable security controls for important services such as online banking or Internet voting.

The existing implementation of Internet voting in Estonia uses pure public key encryption and digital signatures in a double envelope scheme. Since 2013, a mechanism for out of band *cast-as-intended* and *recorded-as-cast* vote verification has been implemented on top of the existing system, introducing a mobile application for verification, independent of the core voting system by design Heiberg and Willemson [18].

The vote is protected by a scheme known as *double envelope*. The scheme defines two independent layers of encapsulation for the vote. For the Estonian implementation, the inner layer is the the value of the vote encrypted with RSA-OAEP using the public key of the voting authority. The outer layer is the signature of the encrypted message, using the signature key of the identification card of the voter. The two layers are processed sequentially by different components of the system. Once all the votes are cast, one server verifies the signature of the votes, removes it from the message and sends the votes, now merely encrypted, to the device in charge of the decryption of the messages. Currently, the decryption device is a Safenet Luna SA HSM.

[2]The possibility of paper voting at a precinct is also available in Estonia, but is considered as an entirely different voting channel.

3.1.2 Need for Stronger Controls

The Estonian Internet Voting system is simple by design. It can be explained to non-technical voters by analogy to voting by physical mail and there are no black box procedures that would generate suspicion. Although it requires a strong technical background, the implementation details, including the source code, are publicly available for audit. However, there are still attack vectors that can be exploited.

One remarkable example is the removal of the two envelopes. In principle, there are two servers srv_1, srv_2. srv_1 receives a batch of votes, each of them of the form $Sig_{voter}(Enc_{authority}(vote))$. It verifies and stripes the signature from the votes. Afterwards, it sends to srv_2 the encrypted votes as a set of elements of the form $Enc_{authority}(vote)$.

If only srv_2 has the secret key $sk_{authority}$, srv_1 can only verify the signatures without learning the value of *vote*. If srv_2 receives only encrypted votes without a signature, it cannot track a specific vote back to a voter.

However, if there is a channel between srv_1 and srv_2, the cooperation between the two servers can break the secrecy of the ballot. A *mix network* can be implemented to mediate the communication between srv_1 and srv_2 while hindering this attack vector.

In a broader sense, a voting scheme needs to employ non-conventional cryptographic techniques to defend against certain attack vectors that cannot be ruled out within a voting scenario. This document focuses on the impact of that fact in the problem of key management.

3.2 Current Approach to Key Management

The current infrastructure of the Estonian Internet Voting system relies on an HSM for decryption, in the role of srv_2. Before the election, the election officials are given hardware tokens. After all the votes are cast, the officials connect the tokens to the HSM and, employing a threshold secret sharing scheme, the votes are decrypted using traditional RSA-OAEP. The secret key is only used during the tallying phase of the process, but the secrecy of the ballot (specifically, the link between the identity of a voter and their vote) must be guaranteed for a long term.

The functionality of the HSM is very well defined, and subject to rigorous certification processes [20, 21]. However, the security guarantees for the key come at the cost of flexibility. As a result, the functionality that can be implemented is very restricted.

On the other hand, for several security critical processes the cost of acquisition and maintenance of an HSM can be hard to justify. Even more so when the process produces no income to cover its operational costs.

The search of more cost effective alternatives to protect cryptographic keys for online voting schemes is the main motivation for the present work.

3.3 Scale in Estonian Internet Voting

The size of a nation-wide election is an important factor, even though Estonian population is relatively small. In particular, the number of candidates and voters may vary significantly between elections. Local elections often feature more candidates, whereas at a country level the amount of voters is the predominant value.

8 elections have been held in Estonia supporting Internet Voting as a voting channel. The last 5 have registered at least 100.000 Internet votes each. The elections for European Parliament in 2015 in Estonia listed 88 candidates. There were almost 900.000 eligible

voters, of which nearly 177.000 cast their votes online [22]. During the local elections of 2013, for the Tartu City Council election alone the number of candidates went up as high as 437 [23].

4 Solution Design

The scenario of online voting introduces a series of trade-offs. Between security and enfranchisement, between different security requirements, between security and efficiency, and so on. Since security is often one of the most critical and ubiquitous requirements in dispute, this section aims to define parameters to enable a comparison between key management strategies. After defining these parameters, three solutions for key management in online voting scenarios are proposed.

As described in section 3.1.1, the Estonian Internet Voting system relies on a double envelope model, where the innermost envelope, which contains the encrypted-only vote, is decrypted by an HSM. The following table presents the main security features of the current HSM-based approach to key management in the Estonian Internet Voting System, as well as additional features that can strengthen the security offered by the system:

Parameter	**Description**
HSM functionality[3]	
Asymmetric algorithms	Batch decryption of votes, currently using RSA. Support for standard ElGamal
Random Number Generation	FIPS 140-2 approved DRBG (SP 800-90 CTR mode)
Level of tamper controls	Tamper evident, tamper resistant
Security certifications	FIPS 140-2 Level 3 [20], Common Criteria EAL 4 (augmented) [21]
MTBF[4]	66561 hours
Access control	Threshold USB tokens
Scalability	The scheme is able to support a nation-wide voting process for the Estonian system, accounting for its consistently growing adoption (see section 3.3)
Desired additional functionality	
Homomorphic encryption support	Additive homomorphic encryption protocols relying on standard security assumptions (e.g. Lifted ElGamal, Paillier, Damgård-Jurik)
Zero Knowledge Proofs	Sigma protocols

4.1 Key Management Quality Assessment

The confidentiality of a cryptographic key is harder to formalize than the confidentiality of raw data because it is at the end of the chain of assumptions. An encryption algorithm, for instance, is secure *under a particular definition* if a set of conditions hold. One of

[3]According to the official functionality described in [24]

[4]Mean Time Between Failures. Standard availability measure

them is the confidentiality of the key. If the key was treated as raw data and encrypted, then there would be another key that would need protection itself, just propagating the problem.

As a result, the comparison between key management strategies requires the definition of its potentially relevant properties. There are good practices defined for certain scenarios, but there is no mature framework to compare different features. This section will propose a definition of such a framework, at least within the scope of cryptographic protocols for online voting.

4.1.1 Trusted Computing Base

In information security the term *Trusted Computing Base* (TCB) refers to a set of components of a system that need to be assumed as flawless, both in terms of functionality and security, in order to be able to build a security model on top of it. It is inherent to the problem: if not even the basic operations of a processor can be trusted, it is impossible to verify that the system implemented actually behaves as it was designed.

For the purpose of this document, the hardware designed specifically for cryptographic functionality (i.e. TPMs, HSMs, smart cards) will be assumed to be within the scope of the TCB. The operations supported by these devices are assumed to work exactly as described and the storage space can only be accessed by an authorized party.

The scope within which the trustworthiness of said devices can actually be held depends on the specific device, and can only be supported by means of a thorough evaluation. For example, the standard FIPS 140-2 defines requirements for cryptographic modules with extensive coverage of tamper protections: evidence, resistance and reaction. There is an extensive offer of cryptographic devices certified at some level defined by FIPS 140-2.

In particular, the implementation of cryptographic functionality within said cryptographic hardware is expected to be designed to resist against known side channel attacks.

4.1.2 Lifecycle of the Key

The attack surface around the key is determined by its lifecycle. By characterizing the generation, transmission, storage, usage and destruction of the key, it is possible to describe more accurately the ways in which the key can be retrieved, and therefore identify how it could be exploited by an adversary. The following criteria are proposed to define the lifecycle of a cryptographic key:

Phase	Feature	Description	Attribute
Key generation	Random number generation	The key is generated using as a parameter the randomness generated by a CSPRNG or a true random number generator.	Confidentiality
Key generation	Verifiable correctness of generator	The key is generated according to the parameters of the algorithm and the correct generation can be attested.	Confidentiality

Transport	Encrypted transport	Any communication of sensitive material (i.e. key or randomness) beyond the boundaries of the TCB.	Confidentiality, Integrity
Usage	Access control	It is possible to define roles and authorized actions around the key. The rules can be provably enforced.	Confidentiality
Usage	The key cannot be used by an unauthorized actor or group	No operation (e.g. encryption, decryption) can use the key if it is triggered by an unauthorized actor or group.	Confidentiality
Storage	Tampering protection	The key cannot be modified (i.e. replaced by another key or destroyed) by an unauthorized actor or group.	Integrity
Storage	Resilience	The key can be used after the failure or non-availability of parts of the TCB.	Integrity
Key destruction	The key can be destroyed irreversibly	It must not be possible to use or retrieve any copy of the key.	Confidentiality

The main idea of these criteria is to remark that obtaining the key is not the only way to break its security. Any aspect of the scheme that leads to calculate the key, non-negligibly reduce the key space or produce a valid encryption is equally critical to the security of the system and needs to be prevented.

Within the scope of this document, this framework is qualitative. The security of the key management strategy can be described in terms of the parameters presented, but no quantitative measure of security is produced from it.

4.2 Design Option 1: Replacing the HSM with Smart Cards

Summary: While the process remains unchanged, the decryption of the votes is supported by smart cards instead of an HSM. A threshold decryption scheme would enhance the decentralization of the model.
Smart card model: Feitian JavaCOS A22
Technology: Java Card 2.2.2
Description: there is a breach between the models for usage and security of an HSM and the ones of a smart card. HSMs are mechanisms designed to be a centralized system that can be accessed by different users with different clearance levels. As a result, they support strong access control techniques, such as threshold secret sharing. On the other hand, smart cards are intended for personal usage and possession is already assumed to be a proof of authorization. In some cases additional controls such as PIN codes are in

place, but they do not support any kind threshold schemes natively. Threshold schemes are paramount to certify that no single individual has too much power within an election.

In particular, there are threshold decryption schemes (based on VSS with untrusted dealer) for public key encryption systems such as RSA[25], ElGamal[26] and Paillier[27]. One of the simplest schemes involves ElGamal.

As it has been stated previously, cryptographic hardware is very limited with regard to the algorithms supported, and traditional ElGamal is not supported. However, ElGamal relies heavily on the same operation as RSA: $a^b \bmod c$.

This fact enabled objective of building a prototype for that functionality, but hindered its performance. It was possible to implement support for ElGamal as a Java Card Applet, but the performance of this software-based implementation was very low. The time was very stable, but very high. Each exponentiation would require between 615 and 625 ms. A single card performed on average 500 operations in 309 seconds. Given its stable behavior, it is estimated that it can perform nearly 5800 operations within an hour.

In other words, just for the most basic implementation of ElGamal, without enabling any threshold feature, considering the transmission time of the ciphertext, analyzing vulnerability against timing or other side channel attacks, and assuming a very conservative voter turnover rate, 20 cards running in parallel for one hour would be required to decrypt the votes for a single country-wide Estonian election.

Even for the execution of built-in operations, smart cards (and Java Cards in particular) are too slow to support the decryption of a big number of ciphertexts.

Result: It is not scalable enough to fulfill the requirements of a nation-wide election.

4.3 Design option 2: Smart Cards with Homomorphic Tallying

Summary: Preprocess the ciphertexts in a more powerful system using homomorphic tallying techniques. Use smart cards to decrypt only the tally.
Smart card model: Feitian JavaCOS A22
Technology: Java Card 2.2.2, preprocessing server
Description: The main obstacle identified during the design of a solution involving smart cards is the amount of decryption operations that are required. Homomorphic encryption schemes enable *certain* operations (addition, for instance) on ciphertexts without decryption, and, specifically, homomorphic tallying is a common example of how homomorphic addition can be leveraged to produce more complex functionality.

In an *additively homomorphic* encryption system, there exists an operation $\otimes$ such that $E(m_1) \otimes E(m_2) = E(m_1 \oplus m_2)$. The addition can be performed repeatedly to obtain the sum of a set of encrypted values: given a set m of size n, $\otimes_{1 \leq i \leq n} E(m_i) = E(\oplus_{1 \leq i \leq n} m_i)$.

The operation of tallying is not far from the addition of a set. Given a list of candidates $l = \{1..L\}$, a list of votes will be represented by a set v s.t. $v_i \in l$. The outcome of the voting process, $T = \mathit{tally}(v) = \{T_1, ..., T_L\}$ is a vector of size L s.t. $T_i = |\{j | 1 \leq j \leq |v| : v_j = l_i\}|$. In plain words, the tally is a vector with the count of the amount of votes for each candidate.

Given an arbitrary vote v_i and a partial tally $T' = \{T'_1, ..., T'_L\}$, the effect of counting v_i into T' is to increase the value of T'_{v_i} by 1. However, ballot secrecy demands that the value of v_i remains secret. In other words, the vote must be correctly counted *without disclosing how it was counted.*

Homomorphic tallying, first proposed by [28], consists of leveraging the homomorphic properties of an encryption system to enable tallying without compromising ballot secrecy.

A popular approach for achieving this objective is to encode the tally as a single integer and define the vote to modify only a specific portion of that integer. As an example, let $L = 3$ and the maximum amount of votes for a candidate be $M = 1111_2 = 15$. The tally will be a $4 \cdot 3 = 12$-bit long integer initialized in 0:

$$\mathrm{T}=\boxed{0000 \mid 0000 \mid 0000}$$

A vote v will be a bit string as long as T with only one bit on, corresponding to the LSB of the region of the tally that represents a specific candidate. For instance, a vote for the candidate 2 will be encoded as:

$$v = \boxed{0000 \mid 0001 \mid 0000} = 16$$

The addition of both bit arrays will increase the tally by 1:

T=	0000	0000	0000
$v =$	0000	0001	0000
$T + v =$	0000	**0001**	0000

The encoding of the vote can be done efficiently, thus reducing the tallying problem to the homomorphic addition problem, under the assumption of an honest voter. For most realistic voting scenarios, however, that assumption does not hold. As a result, a zero knowledge proof of validity must be calculated for every vote.

As can be seen in the example, there are two variables that determine the size of the tally representation: the amount of candidates (L) and the maximum amount of votes (M) that can be cast. In order to prevent integer overflow incidents which would jeopardize the reliability of the election, the size of T in bits must be:

$$|T| = L * \log_2 M \tag{2}$$

These are the bit lengths for different values of L and M.

M	$log_2 M$ \ L	2	10	16	128	1024
4	2	4	20	32	256	2048
64	6	12	60	96	768	6144
1024	10	20	100	160	1280	10240
16384	14	28	140	224	1792	14336
262144	18	36	180	288	2304	18432
4194304	22	44	220	352	2816	22528
67108864	26	52	260	416	3328	26624

Considering a reasonable growth margin on top of the numbers in section 3.3, for the Estonian scenario $M = 2^{18}$ voters and $L = 2^{10} = 1024$ are sensible dimensions. To support those magnitudes, each vote would need to be encoded as a 18432-bit integer.

The performance overhead induced by this encoding cannot be disregarded, but could be accepted if the operations were performed in plaintext in a powerful device. However, the scheme as a whole is not scalable for several reasons:

1. Homomorphic addition is possible, but not efficient.

2. Common alternatives of homomorphic encryption with high values, such as the Paillier cryptosystem, need to perform operations modulo n^s where n is the maximum size of the plaintext and $s \geq 2$ (note that the s within the exponent is an arbitrary constant, not to be associated with a secret value). Arithmetic operations modulo a 18432^2 =339738624-bits long number are not supported by any readily available cryptographic hardware, and are slow in software.

3. The kind of zero knowledge proofs required to attest the validity of the vote encoding, known as *range ZKP*, are possible [29], but inefficient within a smart card environment. In particular, they become the kind of bulk operation that needs to be avoided for a smart card based system.

Result: the scalability of the scheme is severely limited by the number of candidates L and, to a lesser extent, by the maximum number of votes, specially when verifiability and smart cards are involved.

Disclaimer: during the later stages of this project we learned about an alternative approach to homomorphic tallying named *vector tallying*. Although it still demands heavily computation-intensive correctness proofs, it is intended to be more efficient than the basic homomorphic tallying approach and might be refined into a feasible solution for this problem in the future. The idea is presented in [30] and a recent publication suggests a performance improvement by generating zero knowledge proofs of correct mixing of a predefined vector, as opposed to proofs of correct encryption [31].

4.4 Design Option 3: Verifiable Secret Sharing in Smart Cards

Summary: the private key is generated in a verifiable distributed manner. It is only reconstructed at the time of decryption.
Smart card model: Feitian JavaCOS A22
Technology: Java Card 2.2.2, preprocessing server
Description: the options 1 and 2 suggest that threshold decryption cannot be performed efficiently for the input sizes of the Estonian Internet Voting scenario. A possible result is to reduce the security demands of the smart cards within the general scenario: to implement only a Distributed Key Generation approach.

This scheme considers the possibility of weakening the security definition of the system in order to enable its implementation. Specifically, the secret key is built in a distributed manner, but it needs to be calculated in a centralized manner in order to perform the decryption. At that point, the key is stored in a new trusted container, and its security is limited by the security of the container. This container, in practical terms, could be the computer that reads the smart cards.

A prototype for this functionality was built on the grounds of two previous works that enable two different aspects of DKG: the first one enables an efficient implementation in Java Card with a trusted dealer and the latter discusses how to remove the trusted dealer.

4.4.1 DKG in Java Card: the CRISES' Report

A report published in 2013 by a Spanish research group known as CRISES [32], describes the implementation of a threshold variant of ElGamal "designed in order to implement all sensitive operations securely into the JavaCard". The result requires the existence of a trusted dealer, which is chosen at random amongst the smart cards, but addresses both at a theoretical and at a practical level some of the most recurrent and demanding challenges of implementing threshold cryptography on smart cards.

One of the main elements missing in the Java Card API in order to implement custom cryptographic algorithms is an equivalent to the Java class `BigInteger`, which enables support for arbitrary length integers and efficient modular operations. This class is used extensively in Java cryptographic implementations such as BouncyCastle [16]. As an essential building block, the CRISES group built a Java Card class called `MutableBigInteger` (inspired by a homonymic class in the Java API), which implements most of the arithmetic operations. The support for modular operations is enabled by leveraging the support for RSA operations offered by Java Card.

On top of that implementation, the group introduced the functionality required for their protocol: ElGamal key generation, commitments, secret sharing and ElGamal encryption/decryption. The group published the code of their prototype on GitHub [33].

Protocol overview

The DKG is documented in detail in the chapter 3 of their report and involves seven tasks (the VSS task is divided as each part of it needs to produce commitments). The following description remains informal as a different version of the protocol will be proposed at the end of this chapter:

1. **Election and certification of the electoral board**: defines the trustees that will form the voting authority (i.e. the n shareholders) and the initialization of the smart cards.

2. **Creation of the electoral board**: the information about the other parties is shared. Particularly, the threshold configuration and the digital certificates of the other smart cards are stored in each of the cards.

3. **ElGamal key generation:** all the cards get the public ElGamal parameters. One card is selected randomly as the dealer. The dealer generates an ElGamal key pair (sk, pk).

4. **Verifiable Secret Sharing:** The dealer generates the shares of the secret value, distributes the shares and destroys the value sk.

    (a) **Phase 1 - Coefficients:** generates the coefficients of the Shamir polynomial. Commits to them.

    (b) **Phase 2 - Values of x:** builds the polynomial and defines the output of the function $getX(i)$ for $1 \leq i \leq n$. Commits to the values produced.

    (c) **Phase 3 - Shares:** evaluates the polynomial, producing S. *Securely eliminates sk.* Commits to the shares.

5. **Share distribution**: creates secret authenticated channels between the card of the dealer and each of the other cards using their digital certificates. Sends one share to each of the smart cards. *Securely eliminates* the shares that were transmitted.

6. **Share verification**: each smart card decrypts its share and verifies the commitments associated.

7. **Share signing**: each smart card signs a commitment to its share and makes the commitment public.

The expectation of "secure deletion" of the secret and the shares is the single point of failure that is not acceptable for this scenario. Since there is no way to prove that the deletion occurred, it is possible for the dealer smart card to keep the secret inadvertently, either accidentally (e.g. the card is removed before the deletion is invoked) or intentionally (e.g. the code is tampered to not remove the secret).

Efficiency results

The CRISES' report includes a benchmark of their protocol, covering the operations performed on the smart card. Individually, the operations are slow but considered acceptable: the share generation and distribution is performed within minutes, whereas the decryption is performed in a matter of seconds.

This result is consistent with the one of the *design option 1* presented in section 4.2, in the sense that decryption is not scalable to the level of a batch operation. The report, accordingly, remarks that this system needs to be used in the context of homomorphic or hybrid voting systems, but is not realistic in the scenario of mixnet based e-voting protocols.

4.4.2 Untrusted Dealer

An approach to remove the need of the trusted dealer is presented in [34]. It is based on the ElGamal cryptosystem [35] and a VSS variant of Shamir's Secret Sharing. Once the share generation is finished, the encryption, reconstruction and decryption phases are essentially unchanged.

The key distribution process requires that each authorized party executes the protocol, and uses the data published by the other parties in order to complete it. The protocol definition here is adapted to reflect its execution in a concurrent, potentially distributed environment. Let A be the list of authorized shareholders and A_i denote a unique identifier of the i-th shareholder. The function `broadcast(m)` shares a message m with all the other parties, whereas `send(i,m)` shares the message m via a secret authenticated channel with the shareholder A_i.

The definition takes the parameters p, q, g from the underlying ElGamal scheme and the threshold values t, n. The first step in the process is the key generation:

Algorithm 2: `PedersenDKG.LocalKeyGeneration`

```
input       : $p, q, g$
output      : $x_i, h_i$
executed by: every $A_i \in A$

1 $x_i \xleftarrow{\$} \mathbb{Z}_q$;
2 $h_i \leftarrow g^{x_i}$;
3 $r \leftarrow R$ ; // Set R not specified in the source
4 $C_i \leftarrow$Commitment.Commit$(h_i, r_i)$;
5 $broadcast(C_i)$
```

The key pair generated by A_i must not be influenced by any other value $h_j (i \neq j)$. To ensure that property, each shareholder commits to its own h_i before getting access to h_j. Once each shareholder has generated its commitment, the commitments are open and a shared public key is calculated:

Algorithm 3: `PedersenDKG.SharedPublicKeyGeneration`

```
input       : $h_i, C$
output      : $h$
executed by: every $A_i \in A$

1 broadcast($h_i, r_i$);
2 for $A_j \in A \setminus A_i$ do
3     wait($h_j, r_j$) ; // Wait for the other parties to publish their public
        key values
4     if Commitment.Open($h_j, C_j, r_j$)=0 then
5         stopAndReport($h_j, C_j, r_j$)
6 $h \leftarrow \prod_{j=1}^{n} h_j$;
```

Since all the public shares h_i were broadcast, each actor in A obtains the same public key, which is to be made public also for the voters. The rest of the key generation protocol is focused on the cooperation necessary to calculate shares for the value of $x = \sum_{i=1}^{n} x_i$ such that $h = g^x$ while preserving the $(t, n)-$threshold properties.

For the next step, each authorized party must execute an independent instance of `Shamir.SecretSharing`. The resulting coefficients and shares will be identified by two indexes: $a_{j,i}$ will describe the j-th coefficient of A_i (with the wildcard $a_{*,i}$ to identify the whole set of coefficients of A_i). Likewise, $s_{j,i}$ will describe the j-th share of s_i. Each authorized party continues its procedure by generating shares of their values x_i:

Algorithm 4: `PedersenDKG.SharedSecretKeyGenerationInit`

```
input       : $x_i, g, t, n$
output      : $s_{j,*}$ (the shares received from the other parties)
executed by: every $A_i \in A$

1 $a_{*,i}, S_i \leftarrow$Shamir.SecretSharing($x_i, t, n$);
2 for $j \leftarrow 1$ to $t - 1$ do
3     $F_{i,j} \leftarrow g^{a_{j,i}}$;
4     broadcast($F_{i,j}$);
5 for $j \leftarrow 1$ to $n$ do
6     send(j,$s_{j,i}$) ; // Note that for the step i=j, the share is send to
        itself.
```

The first cycle commits to the coefficients of the secret sharing polynomial. The second cycle sends a share of the secret x_i to each member of A. In other words, A_i has a share of each of the secrets created for the protocol. If t members of A agree, they can reconstruct all of the secrets x_i. The last part of the protocol adds the last layer of verifiability and aggregates the shares in order to require one single secret reconstruction process:

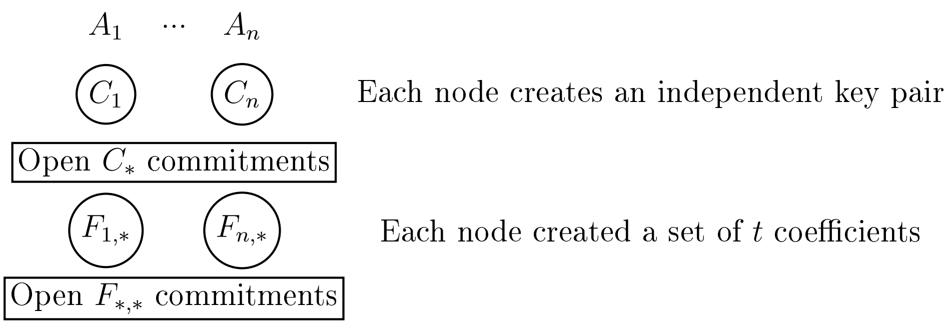
```
Algorithm 5: PedersenDKG.SharedSecretKeyGenerationFinish
   input        : $g, n, s_{i,*}, F, h$
   output       : $s_i$. Additionally signs the computed private key $h$ if the key shares
                  are consistent.
   executed by: every $A_i \in A$
1  for $j \leftarrow 1$ to $n$ do
2      if !verify($g^{s_{i,j}} = \prod_{l=0}^{t-1} F_{j,l}^{il}$) ; // Verify share consistency
3       then
4          reportFailureAndStop();
5  $s_i \leftarrow \sum_{j=1}^{n} s_{j,i}$;
6  publish($Sig_{A_i}(h)$)
```

As originally intended, the result of the process is an ElGamal key pair for which the secret key is $(t, n)-$shared. The reconstruction process of the key is the default `Shamir.SecretReconstruction` scheme, and the encryption and decryption processes are the standard ElGamal algorithms.

The `PedersenDKG` algorithm can be summarized as follows:

```
Algorithm 6: PedersenDKG
   input        : $p, q, g, t, n$
   output       : $s$. Additionally signs the computed private key $h$ if the key shares
                  are consistent.
   executed by: every $A_i \in A$
1  $x_i, h_i \leftarrow$PerdersenDKG.LocalKeyGeneration($p, q, g$);
2  $h \leftarrow$ PerdersenDKG.SharedPublicKeyGeneration($h_i$);
3  $s_{i,*}, F \leftarrow$PerdersenDKG.SharedSecretKeyGenerationInit($x_i, g, t, n$);
4  $s \leftarrow$PerdersenDKG.SharedSecretKeyGenerationFinish($g, n, s_{i,*}, F, h$)
```

Another useful visualization of the algorithm focuses on the use of commitments as synchronization points, as the algorithm requires that the commitments of the other parties are generated (and subsequently correctly open) before continuing its flow:


A_1
$\cdots$
A_n
C_1
C_n
Each node creates an independent key pair
Open C_* commitments
$F_{1,*}$
$F_{n,*}$
Each node created a set of t coefficients
Open $F_{*,*}$ commitments

It is important to remark that the definition of `PedersenDKG.LocalKeyGeneration` uses a generic definition of commitment, whereas the coefficient commitments are explicitly defined as part of the protocol.

4.4.3 Architecture

Although the notation, the entities and the terminology of the CRISES report and the Pedersen's DKG scheme differ vastly, it was possible to modify the CRISES source code to remove the trusted dealer.

A review of the CRISES protocol revealed that the main difference between their implementation and the scheme proposed by Pedersen was the number of users performing the key generation process.

Java Card debugging is a slow low level process. The deployment of a minor change takes minutes and the results are often two byte codes, which might be standardized, application-dependent or both. It is unsuitable to test by itself cryptographic functionality. The resulting decision was the design of a brand new Java application that would communicate with the Java Card only when it was strictly necessary.

The interface `Broker` was designed to represent a party in a threshold protocol. The class `BasicElGamalJavaBroker` extends from it in a simple implementation of traditional ElGamal.

The second implementation, `ThresholdElGamalJavaBroker`, implemented the behavior of an actor A_i within Pedersen's protocol. For the communication between different brokers, a bulletin board was implemented. A bulletin board B is an abstract concept present in online voting protocols that contains all the messages exchanged between the parties. A bulletin board supports the abstract functions `append` and `search`, but not `delete` or `edit`.

Since the nature of the messages in the bulletin board is so diverse, the bulletin board was divided into a set of *subjects*, defined in the Enumeration `BBT`. That way, the functionality of the bulletin board was defined:

```
Algorithm 7: BulletinBoard.addMessage
  input   : BBT tag, Message newMessage
  output  : none. The message newMessage is now attached to the subject
            identified by tag
1 $B[tag] \leftarrow B[tag] \cup \{newMessage\}$
```

```
Algorithm 8: BulletinBoard.getSubject
  input   : BBT tag
  output  : $subject$
1 $subject \leftarrow B[tag]$
```

The reading functionality in B was designed to return all the messages associated with one tag (e.g. all the messages with the tag `PUBLIC_KEY_SHARE_TAG` are to be used together to calculate the public key).

The creation of the bulletin board made it possible to keep track of the communication between the brokers, since no message can be exchanged between them in any other way. Different instances of `Message` were created, out of which `SimpleMessage` and `SecretMessage` are the most important. `SimpleMessage` contains a byte array and `SecretMessage` contains another `Message` that can only be read by an intended broker. Whereas for a real scenario `SecretMessage` would encrypt-and-sign the message, this implementation focused on functionality and offers no real security to the message.

The final phase of the development involved the creation of the class `ThresholdElGamalJavaCardBroker`, which ported the *key generation* part of the `ThresholdElGamalJavaBroker` to the Java Card.

The implementation was tested with a $(3,5)-$threshold where one of the Brokers was a `ThresholdElGamalJavaCardBroker` and the other 4 `ThresholdElGamalJavaBroker`. The implementation succeeded both when the Java Card broker was used to reconstruct the key and when it contributed only to the creation of the secret.

4.4.4 Performance

As described in the previous sections, the key generation protocol implemented is essentially similar to the version developed by the CRISES group. The main difference lies on the amount of cards used to calculate the secret, and since that task can be performed in parallel in a system designed to provide almost constant time performance, the total time of the key generation is not likely to increase. The time for verification of commitments, however, should increase by a factor of n.

Thus, the results provided by the CRISES group for their approach are not likely to increase by several orders of magnitude. Their results for the trusted dealer approach, aggregating the result for different key sizes, were:

Operation	Performance range (minutes)
Generation of 5 shares	5.56 to 20.10
Verification of a share	1.14 to 4.26
Encryption of 1 message	0.42 to 1.25
Decryption of 1 message	0.27 to 0.70

Since the performance overhead will occur during the key generation phase and will not increase by several orders of magnitude, the viability of the process remains intact. The performance-intensive operations can be performed in advance and will not have repercussions during the election process.

The current implementation of the system contains a non-verifiable version of the process. In other words, it does neither calculate nor verify commitments. The non-verifiable share generation process takes in average 95 seconds to generate 5 3-threshold shares, with an error margin of at most 10 seconds.

4.4.5 Solution Analysis

The solutions 1 and 2 were identified as inviable or insufficient before they were evaluated in terms of the framework defined in section 4.1. However, this last solution was implemented and is feasible as long as a non-decryption-intensive voting scheme is introduced. The next step is to analyze the solution from the security point of view.

- *Random number generation:* depends on the smart card. Java Card supports two algorithms, `ALG_PSEUDO_RANDOM` and `ALG_SECURE_RANDOM`. The Feitian Java Card supports both, and the implementation makes use of `ALG_SECURE_RANDOM`.

- *Verifiable correctness of the generator:* the key generation can be certified in some Java Cards for *standard algorithms*. There are no known certifications for smart card environments that certify ElGamal implementations, or threshold implementations in particular.

- *Encrypted transport:* the protocol establishes a secure authenticated channel between TCBs to exchange the secret shares that can be used to decrypt the secret.

- *Access control:* the right to calculate and use a secret share is given in this scheme by the physical possession of the smart card. A PIN code can be used as a second factor of authentication.

- *The key cannot be used by an unauthorized actor or group:* t cards are required to perform a decryption, reducing the impact of the loss of one card.

- *Tampering protection:* a share must be consistent with the published commitments in order to be used, and there are $n - t$ cards that can be used as a replacement if the secret in one of them is tampered with. After the reconstruction, the key is exposed.

- *Resilience:* The system can tolerate up to $n - t$ failures (accidental or intentional) and resist the collusion of groups of at most $t - 1$ authorized users. For a fixed value n of authorized users, the selection of t constitutes a trade-off between confidentiality and resiliency. After the key is reconstructed, if the key is tampered with or destroyed, it can be reconstructed again. However, it can no longer be considered a shared secret, as it existed in its reconstructed state in a less secure container.

  The protocol does not contemplate the creation of additional shares (in case of loss or failure, for instance), but it is possible to design it.

- *The key can be destroyed irreversibly:* the key disposal procedure is not defined by the protocol. The shares should be deleted from the smart cards and the reconstructed copy securely deleted from its trusted container.

Result: assuming a correct implementation and the selection of a reliable Java Card provider with the right supported features, the scheme can resist the collusion of up to $t-1$ authorized parties and the loss of up to $n-t$ shares. After the key *reconstruction*, however, the security of the scheme is limited to the security of the trusted container used to calculate the secret key.

Additional security controls should be added to harden access control, resilience and the secure deletion of the key.

4.5 Result Comparison

As the design options 1 and 2 were found to be inviable under the conditions established, it is only worthwhile to compare the design option 3 against the current HSM approach. The official characteristics of the HSM were summarized at the beginning of this chapter, whereas the security analysis of the design option 3 was the subject of section 4.4.5. Since the HSM results are described in terms of the functionality of the hardware and the suggested option in terms of the key management framework defined in section 4.1, the following section will summarize the effects of migrating from an HSM to a smart card based approach.

4.5.1 Unchanged Features

HSMs and smart cards are both hardware implementations designed to execute cryptographic operations within a hardened environment. It is possible to acquire hardware with similar FIPS 140-2 and Common Criteria security certifications for both systems, and both are used to support security-critical processes. Likewise, although the set of specific algorithms is vendor-dependent, the standardization and compatibility of cryptographic algorithms is high.

4.5.2 New Features

The Java Card scheme introduces a stronger variant of secret sharing that reduces the reliance on a single point of failure, although it does not remove it entirely. The technology enables the generation of zero knowledge proofs as well as the implementation of homomorphic encryption algorithms.

The overall cost of the process is reduced and the resilience of the process is preserved, if not strengthened, due to the nature of the threshold protocol. Nevertheless, the most significant feature achieved is flexibility: the possibility of implementing and executing arbitrary (if limited) protocols is of utmost importance in an environment where the majority of the most powerful protocols are not readily available.

4.5.3 Impacted Features

The main feature affected by the Java Card alternative is the performance of cryptographic operations. As a result, the variety of voting schemes that can be supported at a nation-wide scale is severely limited.

Other features that can be impacted are tamper evidence and availability, as HSMs are designed to be heavy duty hardware and smart cards are not.

5 Conclusion and Future Work

This document described the analysis of alternative key management strategies suitable for the scenario of nation-wide legally binding online voting. The alternatives evaluated aimed at the use of smart cards as a low cost *secure* provider of cryptographic functionalities and key storage. A qualitative framework to evaluate key storage strategies was defined.

While early results already revealed that cryptographic batch operations are not feasible on the existent smart card hardware, the work of the CRISES group in 2013 had proven that an implementation of threshold decryption in smart cards is possible. This document described how that work was extended to remove the figure of the trusted dealer, although it still needs a trusted party to calculate the secret.

Despite the promising result from the functionality point of view, there is a wide range of open questions that need to be addressed before their results can be considered for a practical usage. Some of them build upon the results of the present document, whereas some constitute orthogonal directions towards the final objective. The final paragraphs of this document will attempt to identify and classify them.

5.1 Non-standard cryptography and smart card development

Security certifications can be misleading. A smart card certified on FIPS 140-2 may be designed to be resistant against side channel attacks, but only *for the supported algorithms*. The software implementation of unsupported cryptographic primitives may be ignoring the assumptions on which the security of the smart card is built, and therefore enabling subtle and dangerous vulnerabilities.

Furthermore, cryptography is a field where peer review is essential to prevent minor bugs from becoming catastrophic vulnerabilities. However, there are no standardization attempts or even notation agreements when it comes to functionality for zero knowledge proofs, commitments and other primitives essential to MPC. This is true even for major programming languages (where there are a few unmaintained projects), and specially critical for low level implementations. If every implementation of these protocols must begin by designing how to encode a share, the applicability of the research in this field will progress at a very slow rate.

This call for building blocks is even more urgent in the case of Java Card, where even arithmetic operations must be redefined before implementing any actual functionality. In fact, the suitability of Java Card as a development platform for cryptographic functionality should be evaluated. Should the cost in terms of low level thinking, slow coding rate, learning curve, cryptic debugging and uncertain security expectations be considered unacceptable for productive scenarios? If that is the case, what would be the alternative? Is the design of a smart card with native support for these operations a viable alternative?

5.2 Threshold decryption in online voting scenarios

The security assumption of the voting protocol was weakened in this project to the point where the key is decentralized *until* it needs to be calculated for decryption. Since a scheme with no trusted dealer was implemented, the hardening of that assumption can be attempted, for instance by introducing the idea of threshold decryption. Is it possible to design a voting scheme that satisfies the following requirements?

1. Distributed Key Generation with untrusted dealer
2. Threshold decryption (i.e. without key reconstruction)
3. Execution of critical operations directly on smart cards (or equivalently secure hardware)
4. Practical for a nation-wide election process (from the results of this document, this implies that it must not be decryption-intensive)
5. Efficiently verifiable

5.3 The general key management problem

Since key management is mostly a qualitative information security concern at the core of the eminently mathematical discipline of cryptography, is it possible to quantify the quality of an information security approach? Can the parameters introduced in the qualitative framework in this document be weighed to bear different relevance under different cryptographic scenarios?

References

[1] Dan Goodin. *Neutered random number generator let man rig million dollar lotteries.* 2016. URL: http://arstechnica.com/security/2016/04/neutered-random-number-generator-let-man-rig-million-dollar-lotteries/ (visited on 11/05/2016).

[2] Thomas Hühn. *Myths about /dev/urandom.* 2015. URL: http://www.2uo.de/myths-about-urandom/ (visited on 11/05/2016).

[3] Arjen Lenstra et al. *Ron was wrong, Whit is right.* Tech. rep. IACR, 2012.

[4] Zakir Durumeric et al. 'The matter of heartbleed'. In: *Proceedings of the 2014 Conference on Internet Measurement Conference.* ACM. 2014, pp. 475–488.

[5] K. Ed. Moriarty et al. *PKCS 12: Personal Information Exchange Syntax v1.1.* RFC 7292. July 2014, pp. 1–29. URL: https://tools.ietf.org/html/rfc7292.

[6] Peter Gutmann. 'Lessons Learned in Implementing and Deploying Crypto Software'. In: *The USENIX Association* (Aug. 2002).

[7] ISO/IEC. *Trusted Platform Module.* ISO/IEC ISO/IEC 11889:2009. 2009.

[8] ISO/IEC. *Identification cards - Integrated circuit cards.* ISO/IEC ISO/IEC 7816. 2003.

[9] PUB FIPS. '140-2'. In: *Security Requirements for Cryptographic Modules* 25 (2001).

[10] Adi Shamir. 'How to share a secret'. In: *Communications of the ACM 22 (11)* (1979).

[11] Benny Chor et al. 'Verifiable secret sharing and achieving simultaneity in the presence of faults'. In: *Foundations of Computer Science, 1985., 26th Annual Symposium on.* IEEE. 1985, pp. 383–395.

[12] Berry Schoenmakers. 'A Simple Publicly Verifiable Secret Sharing Scheme and its Application to Electronic Voting'. In: *Advances in Cryptology-CRYPTO '99, Vol. 1666* (1999).

[13] Stephan Neumann, Jurlind Budurushi and Melanie Volkamer. 'Analysis of Security and Cryptographic Approaches to Provide Secret and Verifiable Electronic Voting'. In: *Design, Development, and Use of Secure Electronic Voting Systems.* Ed. by Dimitrios Zissis and Dimitrios Lekkas. IGI Global, 2014. Chap. 2, pp. 27–61.

[14] Bruce Schneier, Kathleen Seidel and Saranya Vijayakumar. 'A Worldwide Survey of Encryption Products'. In: *Berkman Center Research Publication* 2016-2 (2016).

[15] shai. *HElib.* Version unknown. 25th Sept. 2015. URL: https://github.com/shaih/HElib.

[16] The Legion of the Bouncy Castle. *The Legion of the Bouncy Castle - Specifications.* 2016. URL: https://bouncycastle.org/specifications.html (visited on 11/02/2016).

[17] OpenSSL Software Foundation. *OpenSSL - ciphers.* 2015. URL: https://www.openssl.org/docs/manmaster/apps/ciphers.html (visited on 11/02/2016).

[18] Sven Heiberg and Jan Willemson. 'Verifiable internet voting in Estonia'. In: *Electronic Voting: Verifying the Vote (EVOTE), 2014 6^{th} International Conference on* (Oct. 2014).

[19] Drew Springall et al. ‘Security Analysis of the Estonian Internet Voting System’. In: *CCS’14: 21st ACM Conference on Computer and Communications Security* (Nov. 2014).

[20] SafeNet. *Level 2 Non-proprietary Security Policy for Luna®PCI-E Cryptographic Module and Luna®PCI-E Cryptographic Module for Luna® SA*. Brochure. 2015. URL: `http://csrc.nist.gov/groups/STM/cmvp/documents/140-1/140sp/140sp2427.pdf`.

[21] TUV Rheinland Nederland B.V. *SafeNet Luna®PCI configured for use in Luna®SA 4.5.1 (RF) with Backup*. Brochure. 2013-07-26. URL: `https://www.commoncriteriaportal.org/files/epfiles/%5BCR%5D%20NSCIB-CC-12-36718-CR.pdf`.

[22] Vabariigi Valimiskomisjon (Estonian National Electoral Committee). *Statistics about Internet Voting in Estonia*. 2015. URL: `http://www.vvk.ee/voting-methods-in-estonia/engindex/statistics/` (visited on 17/05/2016).

[23] Vabariigi Valimiskomisjon (Estonian National Electoral Committee). *(In Estonian) Kohaliku omavalitsuse volikogu valimised 2013*. 2013. URL: `http://info.kov2013.vvk.ee/kandidaadid/?code=0795` (visited on 17/05/2016).

[24] Gemalto. *Gemalto SafeNet Luna SA - Hardware Security Module - Product Brief*. Brochure. 2015. URL: `http://www.safenet-inc.com/WorkArea/linkit.aspx?LinkIdentifier=id&ItemID=8589949133`.

[25] Ivan Damgård and Maciej Koprowski. *Practical threshold RSA signatures without a trusted dealer*. Springer, 2001.

[26] Yvo Desmedt and Yair Frankel. ‘Threshold cryptosystems’. In: *Advances in Cryptology-CRYPTO’89 Proceedings*. Springer. 1989, pp. 307–315.

[27] Takashi Nishide and Kouichi Sakurai. ‘Distributed paillier cryptosystem without trusted dealer’. In: *Information Security Applications*. Springer, 2010, pp. 44–60.

[28] Josh D Cohen and Michael J Fischer. *A robust and verifiable cryptographically secure election scheme*. Yale University. Department of Computer Science, 1985.

[29] Jan Camenisch, Rafik Chaabouni et al. ‘Efficient protocols for set membership and range proofs’. In: *Advances in Cryptology-ASIACRYPT 2008*. Springer, 2008, pp. 234–252.

[30] Aggelos Kiayias and Moti Yung. ‘The Vector-Ballot E-Voting Approach’. In: *Lecture Notes in Computer Science*. Ed. by Ari Juels. Springer, 2004. Chap. Financial Cryptography, pp. 72–89.

[31] Víctor Mateu, Josep M. Miret and Francesc Sebé. ‘A hybrid approach to vector-based homomorphic tallying remote voting’. In: *International Journal of Information Security* (Apr. 2016).

[32] Jordi Pujol-Ahulló et al. *TTP SmartCard-based ElGamal Cryptosystem using Threshold Scheme for Electronic Elections (Extended)*. Tech. rep. 2013.

[33] Jordi Pujol-Ahulló et al. *eVerification2*. Version unknown. 29th Jan. 2013. URL: `https://github.com/CRISES-URV/eVerification-2`.

[34] Torben Pryds Pedersen. ‘A Threshold Cryptosystem without a Trusted Party’. In: *Advances in Cryptology-EUROCRYPT ’91* (1991), pp. 522–526.

[35] Taher ElGamal. 'A public key cryptosystem and a signature scheme based on discrete logarithms'. In: *Advances in cryptology*. Springer. 1984, pp. 10–18.

A Notation Summary

A.1 Basic Notation

Symbol	**Meaning**
$\oplus$	Group addition
$\otimes$	Group multiplication
$\forall$	For all
g	Group generator
p, q	Arbitrary prime numbers
$Pr[x]$	Probability of event x
$\mathbb{Z}_p$	Modular group of size p
$x \leftarrow y$	Assign the value y to the variable x
$x \overset{\$}{\leftarrow} X$	Assign a random value from the set X to the variable x
$X \setminus Y$	The set X without the elements in Y
$x \in X$	The element x is in the set X
$x \notin X$	The element x is NOT in the set X
$X \subset Y$	The set X is a proper subset of the set Y
$X \subseteq Y$	The set X is a subset of the set Y

A.2 Basic Cryptographic Functions

Symbol	**Meaning**
c	Encrypted message
$Enc_A(m)$	The encryption of the message m with the public key of the user A
h	ElGamal public key
k	Generic cryptographic key
K	Key space. Set of possible values of a key
m	Plaintext message
pk	Public key
pk_A	Public key associated to the user A
r	Random value
$Sig_A(m)$	The signature of the message m with the private key of the user A
sk	Secret key
sk_A	Secret key associated to the user A
x	ElGamal private key

A.3 Secret Sharing

Symbol	**Meaning**
A	Authorized shareholders
A_i	i-th authorized shareholder
a_i	i-th polynomial coefficient
$a_{*,i}$	Set of polynomial coefficients of A_i
$a_{j,i}$	j-th polynomial coefficient of A_i
C	Commitment or set of related commitments
n	Total amount of shares generated
t	Threshold. Amount of shares required to perform an operation
s	Secret value
S	Set of n shares of a secret
S_i	Set of shares of a secret generated by A_i
s_i	i-th share, expressed as a point (x_i, y_i)
$s_{j,i}$	j-th share generated by A_i, expressed as a point $(x_{j,i}, y_{j,i})$
x_i	x component of s_i
y_i	y component of s_i

A.4 Online Voting

Symbol	**Meaning**
B	Bulletin board
B_i	Part of the bulletin board corresponding to V_i
l	Set of candidates
L	Amount of candidates
M	Maximum amount of votes possible for a single candidate
T	Tally
V	Ordered set of voters that voted
V_i	Unique id of the i-th voter
v	Ordered set of votes
v_i	Vote of V_i

A.5 Others

Symbol	**Meaning**
srv_i	i-th server in a set of servers

B Acronyms

Acronym	**Meaning**
AES	Advanced Encryption Standard
API	Application Programming Interface
CAST	Carlisle Adams Stafford Tavares
CSPRNG	Cryptographically Secure PRNG
CTR	CounTeR [encryption operation mode]
CVE	Common Vulnerabilities and Exposures
DKG	Distributed Key Generation
DRBG	Deterministic Random Bit Generator
EC	Elliptic Curve
ECDH	EC Diffie-Hellman
ECDSA	EC Digital Signature Algorithm
ECIES	EC Integrated Encryption Scheme
FIPS	Federal Information Processing Standards
HSM	Hardware Security Module
IDEA	International Data Encryption Algorithm
IEC	International Electrotechnical Commission
ISO	International Organization for Standardization
MAC	Message Authentication Code
MD5	Message Digest [Algorithm] 5
MPC	Multi-Party Computation
MTBF	Mean Time Between Failures
OAEP	Optimal Asymmetric Encryption Padding
PKCS	Public Key Cryptography Standards
PRNG	Pseudo-Random Number Generator
RAM	Random Access Memory
RC2/RC4/RC5	Rivest Cipher 2/4/5
RFC	Request For Comments
RSA	Rivest, Shamir, Adleman [cryptosystem]
SHA	Secure Hash Algorithm
SSL	Secure Socket Layer
TCB	Trusted Computing Base
TPM	Trusted Platform Module
VSS	Verifiable Secret Sharing
ZKP	Zero Knowledge Proof

C Basic Building Blocks

This document contains references to several basic cryptographic primitives, such as encryption, digital signature and commitments. Although they are not the focus of this work, their definitions are provided for completeness. The security parameters and security definitions for each of them are unspecified, as they may change for different scenarios in which they are applied.

C.1 Public Key Encryption

A public key encryption scheme `PK` is defined by three algorithms, `PK.KeyGeneration`, `PK.Encryption`, `PK.Decryption`. Specifically:

> `PK.KeyGeneration` produces a key pair (sk, pk), where sk is secret and pk is public.
>
> Given a message m, `PK.Encryption`(pk, m) returns c, an encryption of m that can only be decrypted by the holder of sk.
>
> Conversely, `PK.Decryption`$(sk, c) = m$.

Encryption ensures mainly that only the authorized recipient will read the message. Specific encryption schemes are designed to support additional security properties. In this document, `PK.Encryption` and `PK.Decryption` are abbreviated as $Enc_{pk}(m)$ and $Dec_{sk}(c)$.

C.2 Digital Signature

`PK` can be extended to support integrity by defining two additional algorithms: `PK.Sign` and `PK.Verify`:

> `PK.Sign`(sk, m) produces the signature sig of the hash value of m.
>
> `PK.Verify`(pk, m, sig) returns 1 iff sig is a valid signature of the message m by the owner of sk.

A digital signature is produced to certify the authenticity and integrity of m.

C.3 ElGamal Cryptosystem

ElGamal is a public key cryptosystem based on the discrete logarithm assumption, often used as the basic scheme for more complex schemes (including homomorphic and threshold schemes). It is defined for both prime integer and elliptic curve groups. The integer version is defined as follows (only key generation, encryption and decryption are relevant for the current document. Signature generation and verification is out of scope):

Algorithm 9: `ElGamal.KeyGeneration`

```
input  : $q, g$
output: $sk, pk$
1 $x \xleftarrow{\$} \mathbb{Z}_q$;
2 $h \leftarrow g^x$;
3 $sk, pk \leftarrow x, h$;
```

Algorithm 10: `ElGamal.Encryption`

```
input  : $q, g, pk, m \in \mathbb{Z}_q$
output: $c$
1 $r \xleftarrow{\$} \mathbb{Z}_q$;
2 $c_1 \leftarrow g^r$;
3 $c_2 \leftarrow m \cdot pk^r$;
4 $c \leftarrow (c_1, c_2)$
```

Algorithm 11: `ElGamal.Decryption`

```
input  : $q, g, sk, c$
output: $m$
1 $m \leftarrow c_2 \cdot c_1^{-sk}$
```

C.4 Commitments

In MPC protocols, a threat model known as the *malicious model* assumes that all parties can misbehave by deviating from a protocol. One of these malicious behaviors consists of changing their output according to the output of the other parties. In scenarios where that situation is not desirable, each party must *commit* to a value before they are revealed.

A `Commitment` then has three phases: `Commitment.KeyGeneration` generates a public key pk. `Commitment.Commit`$_{pk}(m, r)$ produces a value C that commits to a message m and takes a random value r. `Commitment.Open`$_{pk}$ takes a tuple (m, C, r) and returns 1 iff $C =$`Commitment.Commit`$_{pk}(m, r)$ and 0 otherwise. In other words, it verifies if the commitment is valid for the claimed value of m.

There are three security properties for a commitment:

- *Completeness:* it is well defined for every possible input.
- *[Computationally] Hiding:* the commitment C does not leak information about the value m to a [polynomial] adversary.
- *Binding:* A commitment cannot be opened for a value $m' \neq m$.