NTNU
Norwegian University of
Science and Technology

Formal verification of the Norwegian Internet Voting Protocol

Solvei Slågedal

Master of Science in Physics and Mathematics
Submission date: June 2018
Supervisor: Kristian Gjøsteen, IMF

Norwegian University of Science and Technology
Department of Mathematical Sciences

Table of Contents

Abstract

In this paper we look at the formalization and verification of several security components of the cryptosystem underlying The Norwegian Internet Voting Protocol. The focus will be on vote submission and B-Integrity.

Sammendrag

I denne artikkelen ser vi på formalisering og verifisering av diverse sikkerhetskomponenter av det underliggende kryptosystemet til den norske e-valgprotokollen. Fokuset vil være på stemmeavgivning og B-integritet.

Chapter 1

Introduction

In this paper, the cryptosystem underlying The Norwegian Internet Voting Protocol will be presented, analyzed and some components of it formally verified using the computer-aided cryptographic proof framework EASYCRYPT.

The players will be

- V The voter
- P The voter's computer
- B The ballot box
- R The return code generator
- D The decryptor
- A The auditor.

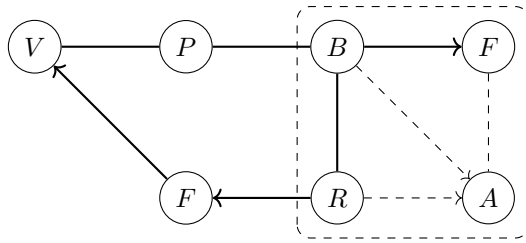


Figure 1.1: Overview of the protocol players and communication channels. [ste13]

1.0.1 Abbrevations

ZK	=	Zero Knowledge
SHVZK	=	Special Honest Verifier Zero Knowledge
NIZK	=	Non-interactive Zero Knowledge
ROM	=	Random Oracle Model
PRF	=	Pseudo Random Function Family
DDH	=	Decision Diffie-Hellman

1.0.2 Notations

In the following paper, the following notations are used.

A dollar sign \$ over an arrow, $x \xleftarrow{\$} X$, is used to denote that x is sampled at random from the distribution X.

A distribution over X is denoted dX.

A vector $(v_1, ..., v_k)$ is denoted as $\vec{v}$.

Let $\mathcal{A}^{\mathcal{O}}$ denote that the algorithm $\mathcal{A}$ has access to an oracle $\mathcal{O}$.

Chapter 2

Theoretical background

To argument for the security in components of the cryptosystem the voting protocol is build upon, some properties will be needed. In this chapter the theoretical background for this will be introduced.

The cryptosystem uses ElGamal encryption (§2.4) and the fact that ElGamal is homomorphic (§2.3) with regards to multiplication.

For the cryptosystem and security, there will be a group structure (§2.2) with certain properties and related problems, among then the Decision Diffie-Hellman (§2.5) distinguishing problem (§2.1).

For the security in the return code generator, a hybrid argument (§2.6) is used.

In a simplified version of the protocol, security in the ballot box is argued with the Decision Diffie-Hellman assumption. An encoding function from a pseudo-random function family (§2.7) is used for the encryption of the ballots, and this concept is explained here.

To assure that certain computations have been done correctly, some non-interactive Zero Knowledge Proofs (§2.10) will be needed. In this chapter the interactive protocols for generating and verifying such proofs are introduced, along with a transformation (§2.9) to make them non-interactive to argue security in the random oracle model (§2.8).

2.1 Indistinguishability

A *distinguishing problem* P is a *problem* (S, X_0, X_1) where S is a set of *instances*, and X_0 and X_1 are two probability spaces over S. An algorithm can achieve success probability $1/2$ simply by guessing.

A measure on "how good" an algorithm $\mathcal{A}$ is at solving P, called the *advantage*, is

$$\mathrm{Adv}_P(\mathcal{A}) = 2\left|\mathrm{Succ}_P(\mathcal{A}) - \frac{1}{2}\right|,$$

where $\mathrm{Succ}_P(\mathcal{A})$ is the *success probability* of an algorithm $\mathcal{A}$ solving P,

$$
\begin{aligned}
\mathrm{Succ}_P(\mathcal{A}) &= \Pr[\mathcal{A}(x) = b \mid b \xleftarrow{\$} \{0,1\}, x \xleftarrow{\$} X_b] \\
&= \Pr[\mathcal{A}(x) = 1 \mid x \xleftarrow{\$} X_1] \cdot \Pr[b = 1 \mid b \xleftarrow{\$} \{0,1\}] \\
&\quad + \Pr[\mathcal{A}(x) = 0 \mid x \xleftarrow{\$} X_0] \cdot \Pr[b = 0 \mid b \xleftarrow{\$} \{0,1\}] \\
&= \Pr[\mathcal{A}(x) = 1 \mid x \xleftarrow{\$} X_1] \cdot \frac{1}{2} + \Pr[\mathcal{A}(x) = 0 \mid x \xleftarrow{\$} X_0] \cdot \frac{1}{2} \\
&= \Pr[\mathcal{A}(x) = 1 \mid x \xleftarrow{\$} X_1] \cdot \frac{1}{2} + (1 - \Pr[\mathcal{A}(x) = 1 \mid x \xleftarrow{\$} X_0]) \cdot \frac{1}{2}.
\end{aligned}
$$

Then the advantage becomes

$$
\mathrm{Adv}_P(\mathcal{A}) = \left| \Pr[\mathcal{A}(x) = 1 \mid x \xleftarrow{\$} X_1] - \Pr[\mathcal{A}(x) = 1 \mid x \xleftarrow{\$} X_0] \right|.
$$

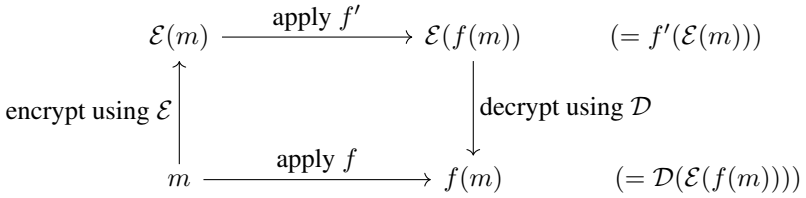
2.2 The Group Structure

To build the cryptosystem (and the voting protocol on top of it), we need a group structure with some specific properties needed to be able to compress ElGamal ciphertexts. In this paper only the properties needed in the case study will be given, but full descriptions can be found in [ste13].

Let q be a prime such that $p = 2q + 1$ is also a prime. The group G which is going to be used in this paper, is the finite cyclic group of quadratic residues modulo p. The group will have prime order q and a generator g.

2.3 Homomorphic encryption

Homomorphic encryption is a method that allows operations to be done to ciphertexts instead of the plaintext, and which yields the same result when decrypted. The point is to be able to perfom operations on encrypted data without needing to decrypt it. An encryption scheme can be homomorphic in regards to some particular operations but not to others. Then the scheme is called homomorphic with regards to those operations, and is called a partially homomorphic cryptosystem. Schemes that are homomorphic in regards to arbitrary operations are called fully homomorphic.



2.4 Multi-ElGamal

Let G be a cyclic group of prime order q with generator g.

ElGamal

$\mathcal{K}$:

$$sk \xleftarrow{\$} \{1, ..., q-1\}$$
$$(pk, sk) \leftarrow (g^{sk}, sk)$$

$\mathcal{E}(pk, m)$:

$$y \xleftarrow{\$} \{1, ..., q-1\}$$
$$c \leftarrow (g^y, pk^y \cdot m)$$

$\mathcal{D}(sk, c)$:

$$(c_1, c_2) \leftarrow c$$
$$m \leftarrow (c_1 \cdot c_2^{-sk})$$

Multi-ElGamal

$\mathcal{K}$:

$$sk_i \xleftarrow{\$} \{1, ..., q-1\},\ 1 \leq i \leq k$$
$$sk \leftarrow (sk_1, ..., sk_k)$$
$$pk \leftarrow (g^{sk_1}, ..., g^{sk_k})$$

$\mathcal{E}(pk, (m_1, ..., m_k))$:

$$y \xleftarrow{\$} \{1, ..., q-1\}$$
$$c_i \leftarrow (g^y, pk_i^y \cdot m_i)$$
$$c \leftarrow (c_1, ..., c_k)$$

$\mathcal{D}(sk, c)$:

$$(c_1, c_{2i}) \leftarrow c_i$$
$$m_i \leftarrow (c_{2i} \cdot c_1^{-sk_i})$$
$$m \leftarrow (m_1, ..., m_k)$$

To the left is the standard ElGamal encryption scheme, and to the right is a multi-variant with k messages and k corresponding keys.

ElGamal is homomorphic in regard to multiplication. That is, for two plaintexts m_1 and m_2,

$$\begin{aligned}\mathcal{E}(m_1) \cdot \mathcal{E}(m_2) &= (g^{y_1}, pk^{y_1} m_1) \cdot (g^{y_2}, pk^{y_2} m_2) \\ &= (g^{y_1+y_2}, (m_1 \cdot m_2) pk^{y_1+y_2}) \\ &= \mathcal{E}(m_1 \cdot m_2).\end{aligned}$$

Then $\mathcal{D}(\mathcal{E}(m_1) \cdot \mathcal{E}(m_2)) = \mathcal{D}(\mathcal{E}(m_1 \cdot m_2)) = m_1 \cdot m_2$.

2.5 The Decision Diffie-Hellman problem

Given a multiplicative cyclic group G with generator g, order q, and a set of integers $\mathbb{Z}_q = \{0, ..., q-1\}$, the Decision Diffie-Hellman problem is to distinguish between the Diffie-Hellman tuple outputted by **DDH0** and the random tuple outputted by **DDH1**.

DDH0

$$a \xleftarrow{\$} \mathbb{Z}_q$$
$$u \leftarrow g^a$$
$$v \xleftarrow{\$} G$$
$$w \leftarrow v^a$$

Diffie-Hellman tuple (g, u, v, w).

DDH1

$$a \xleftarrow{\$} \mathbb{Z}_q$$
$$u \leftarrow g^a$$
$$v \xleftarrow{\$} G$$
$$w \xleftarrow{\$} G$$

Random tuple (g, u, v, w).

In both games a is sampled from $\mathbb{Z}_q$ and v from G, and $u = g^a$. In **DDH0**, $w = v^a$, and in **DDH1**, w is sampled from G. The challenge is in determining whether $w = v^a$ or random. The Decision Diffie-Hellman assumption states that, given $u = g^a$ and v uniformly and independently chosen, w "looks like" v^a. That is, the probability distributions of the two tuples are computationally indistinguishable. Distinguishing a random w from $w = v^a$ is considered a hard problem.

Let X_1 be the set of random tuples and X_0 the set of DDH-tuples, then the advantage of a DDH-distinguisher is

$$\mathrm{Adv}_{\mathrm{DDH}}(\mathcal{D}) = \left|\Pr[\mathcal{D}(g,u,v,w) = 1 \mid (u,v,w) \xleftarrow{\$} X_1] - \Pr[\mathcal{D}(g,u,v,w) = 1 \mid (u,v,w) \xleftarrow{\$} X_0]\right|.$$

2.6 Hybrid argument

We want to compare the two distributions D_0 and D_1. Let $D_0 := H_0, H_1, ..., H_n =: D_1$ be a sequence of hybrid distributions where H_i can be seen as an interpolation from H_0 to H_n where the distances between the adjacent H_i's are small.

Then $f(D_0) - f(D_1)$ can be written as a telescoping series

$$f(D_0) - f(D_1) = f(H_0) - f(H_n) = \sum_{i=0}^{n-1} \left(f(H_i) - f(H_{i+1})\right),$$

and bounding $f(D_0) - f(D_1)$ may be reduced to bounding the terms in the sum.

Let the advantage of an algorithm $\mathcal{A}$ be denoted by

$$\mathrm{Adv}^{\mathrm{dist}}_{H_i,H_{i+1}}(\mathcal{A}) = \left|\Pr[\mathcal{A}(x) = 1 \mid x \xleftarrow{\$} H_i] - \Pr[\mathcal{A}(x) = 1 \mid x \xleftarrow{\$} H_{i+1}]\right|,$$

then, by the triangle equality, we have that

$$\mathrm{Adv}^{\mathrm{dist}}_{D_0,D_1}(\mathcal{A}) \leq \sum_{i=0}^{n-1} \mathrm{Adv}^{\mathrm{dist}}_{H_i,H_{i+1}}(\mathcal{A}),$$

and

$$\mathrm{Adv}^{\mathrm{dist}}_{D_0,D_1}(\mathcal{A}) \leq n\mathrm{Adv}^{\mathrm{dist}}_{H_k,H_{k+1}}(\mathcal{A})$$

for some $k,\ 0 \leq k < n$.

2.7 Pseudo-Random Function families

Pseudo-Random Function (PRF) families are functions $F : K \times D \to R$ where K is a keyspace equipped with a distribution dK, D is the domain, and R is the range equipped with a distribution dR. The keys are sampled from dK. Let PRF-security denote the indistinguishability between a random function (defined by dR, and with key sampled from dK) and a function from the PRF family with key sampled from dK. The function from the family will be used in game **PRFr** (real), and the random function will be used in **PRFi** (ideal), below.

PRFr

On input x,

$$k \xleftarrow{\$} dK$$
$$r \leftarrow F(k, x)$$
$$b \leftarrow \mathcal{D}(r)$$

PRFi

On input x,

$$r \xleftarrow{\$} dR$$
$$b \leftarrow \mathcal{D}(r)$$

Let X_0 be the range of the function from the function family in **PRFr** and X_1 the range of the random function in **PRFi,** then the advantage of a PRF-distinguisher $\mathcal{D}$ is

$$\text{Adv}_F^{\text{PRF}}(\mathcal{D}) = \left|\Pr[\mathcal{D}(r) = 1 \mid r \xleftarrow{\$} X_0)] - \Pr[\mathcal{D}(r) = 1 \mid r \xleftarrow{\$} X_1]\right|.$$

2.8 Random Oracle Model

The Random Oracle Model is a model in which cryptographic hash functions are modelled as random oracles. A random oracle is an oracle that responds to every unique query with a response sampled uniformly at random from its output space, and, if a query is repeated, it repeats the response corresponding to that query. A sketch of a possible implementation is given below.

ROM $\mathcal{H}$

on input i

$$y \xleftarrow{\$} dH$$
$$H(i) \leftarrow \begin{cases} y, i \notin domH \\ H(i), i \in domH \end{cases}$$

output $H(i)$

On query i, the random oracle $\mathcal{H}$ will check if i already is in the domain of the recorded queries to H, and if so, return $H(i)$. If i is not in its domain, it will set $H(i)$ equal to the randomly sampled y from the output distribution dH and return it.

2.9 Fiat-Shamir transformation

Let H be a hash function in the random oracle model. The prover will compute a value h by obtaining it from the hash function on an input x. Then the verifier will obtain a value h' from H on input x', where x' is some input calculated from x and the proof the prover has outputted. The verifier may now check whether $h' = h$, and accept if it is, or reject if it is not.

2.10 Non-interactive Zero Knowledge Proofs

At some points we will need to know that certain computations have been done correctly, and this is done by Non-Interactive Zero Knowledge (NIZK) proofs. This will force an adversary to prove that she knows the content of a ciphertext, and that certain computations have been done correctly.

2.10.1 Σ-Protocols

Let a prover $\mathcal{P}$ and a verifier $\mathcal{V}$ have common input s. $\mathcal{P}$ sends $\mathcal{V}$ a message a, then $\mathcal{V}$ sends $\mathcal{P}$ a random reply b. Lastly $\mathcal{P}$ sends a reply c, and $\mathcal{V}$ decides to accept or reject based on the data it has seen (s, a, b, c).

A protocol P is a Σ-protocol for relation R if

- P is on the above 3-move form, and have completeness.
- Satisfies the special soundness property. That is, from any statement s and corresponding accepting conversations (a, b, c) and (a, b', c'), where $b \neq b'$, one can compute w such that $(s, w) \in R$.
- Satisfies the Special Honest Verifier Zero Knowledge (SHVZK) property. That is, there exists a polynomial-time simulator $\mathcal{S}$, which gets as input s and a random b and outputs an accepting conversation (a, b, c) with the same probability distribution as conversations between the honest prover and verifier on input s.

2.10.2 Equality of Discrete Logarithms

The voter's computer P uses the encryption algorithm $\mathcal{E}$. We want to make sure that P computes correctly during the encryption. For this, we will use a proof that two elements have the same discrete logarithm relative to two distinct generators of the group G. The proof is $\pi_{\mathcal{E}}$, and is between a prover $\mathcal{P}^I_{eqdl}$ and a verifier $\mathcal{V}^I_{eqdl}$.

The ballot box B uses the transformation algorithm $\mathcal{T}$. We also want to make sure that B computes correctly. For this, we will need two equality of discrete logarithms proofs.

One that proves that several group elements are raised to the same power as a certain element, and one that proves that several elements has been correctly raised to distinct powers. The proofs are $\pi_{\mathcal{T}I}$ and $\pi_{\mathcal{T}II}$, respectively, and are between a prover $\mathcal{P}^{II}_{eqdl}$ and a verifier $\mathcal{V}^{II}_{eqdl}$, and a prover $\mathcal{P}^{III}_{eqdl}$ and a verifier $\mathcal{V}^{III}_{eqdl}$, respectively.

The three proofs above, must all satisfy the three properties *Completeness, SHVZK,* and *Soundness* with definitions given below.

Completeness Any proof generated by an honest $\mathcal{P}$ must be accepted by $\mathcal{V}$.

Special Honest Verifier Zero Knowledge That the proof is SHVZK means that on a given challenge e, a simulator should be able to choose a random ν which produces transcripts indistinguishable from the real ones. The simulator will not get the private input w.

Soundness Let $Rel(s, w)$ be the relation between the statement s and the witness w. It must be hard to generate valid proofs when the relation does not hold.

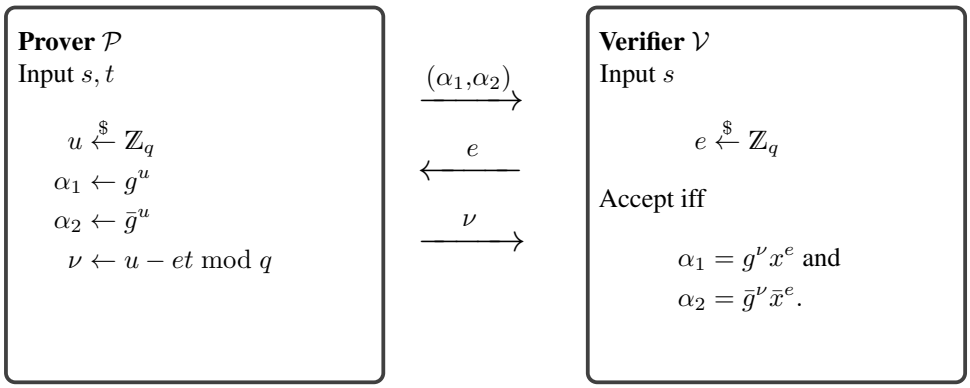
Proof of correct computation I

We want to prove that two element have the same discrete logarithm relative to distinct generators of the group G. Both the prover and the verifier are given as public input some auxiliary information aux, two generators g and $\bar{g}$ of G, and the commitments $x, \bar{x}$. The prover is in addition given the private input integer t. The relation is $Rel(x, t) = (\log_g x \stackrel{?}{=} \log_{\bar{g}} \bar{x} \stackrel{?}{=} t)$.

For this equality of discrete logarithms problem, the prover and the verifier algorithms are:

$$\pi_{\mathcal{E}} \leftarrow \mathcal{P}^{I}_{eqdl}(aux, t; g, \bar{g}; x, \bar{x})$$
$$1/0 \leftarrow \mathcal{V}^{I}_{eqdl}(aux; g, \bar{g}; x, \bar{x}; \pi_{\mathcal{E}}).$$

Instantiation A Σ_3-protocol for the Equality of Discrete Logarithms with prover $\mathcal{P}$, verifier $\mathcal{V}$, statement $s = (aux, g, \bar{g}, x, \bar{x})$ and witness t such that $x = g^t$ and $\bar{x} = \bar{g}^t$ is:



The completeness condition requires that any proof generated by an honest $\mathcal{P}$ must be accepted by $\mathcal{V}$.

Applying the Fiat-Shamir transformation we get the following.

Let $H_1 : \{0,1\}^* \times G^6 \to \{1, 2, ..., 2^\tau\}$ be a hash function in the random oracle model. Then the prover will compute

$$e \leftarrow H_1(aux, g, \bar{g}, x, \bar{x}, \alpha_1, \alpha_2),$$

and the verifier will compute

$$e' \leftarrow H_1(aux, g, \bar{g}, x, \bar{x}, g^\nu x^e, \bar{g}^\nu \bar{x}^e).$$

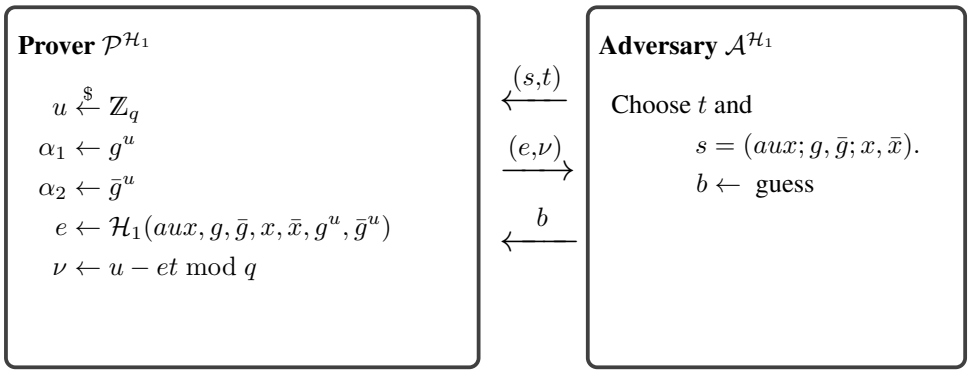
The proof is $\pi_\mathcal{E} = (e, \nu)$, and the verifier will accept iff $e' = e$.

Security analysis Two properties for the proofs are required: they most be zero knowledge and satisfy a soundness condition. For the zero knowledge it suffices for the protocol to be Special Honest Verifier Zero Knowledge (SHVZK), the non-interactive proof will then be Non-interactive Zero Knowledge, with the hash oracle acting as an honest verifier. For the Soundness property, there need to exist a simulator that on a given input produces a conversation with the same probability as in the real proof.

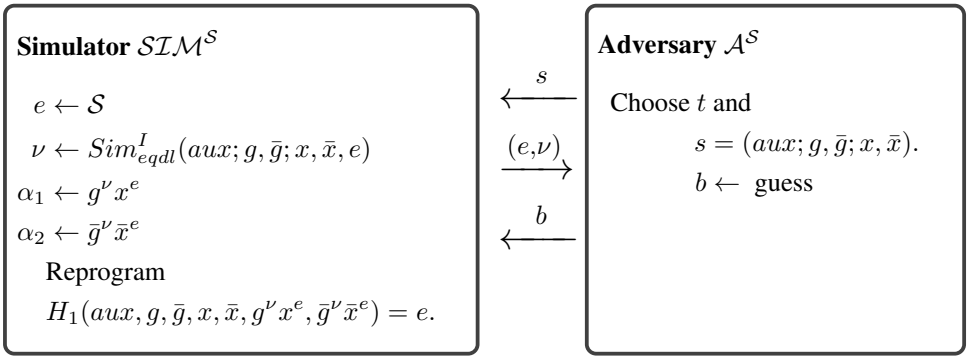
Completeness An honest $\mathcal{P}$ will on statement $(aux, g, \bar{g}, x, \bar{x})$ generate a proof $(e, \nu) = (H_1(aux, g, \bar{g}, x, \bar{x}, g^u, \bar{g}^u), u - et \bmod q)$. The verifier $\mathcal{V}$ will then compute $e' = H_1(aux, g, \bar{g}, x, \bar{x}, g^\nu x^e, \bar{g}^\nu \bar{x}^e)$. It is easy to see that $e' = e$ if the relation holds, and hence the proof will be accepted by $\mathcal{V}$.

Special Honest Verifier Zero Knowledge Given any challenge e, we can choose a random ν and compute $\alpha_1 = g^\nu x^e$ and $\alpha_2 = \bar{g}^\nu \bar{x}^e$, with $x = g^t$ and $\bar{x} = \bar{g}^t$. It is easy to see that α_1 and α_2 has the same distribution as in the real proof.

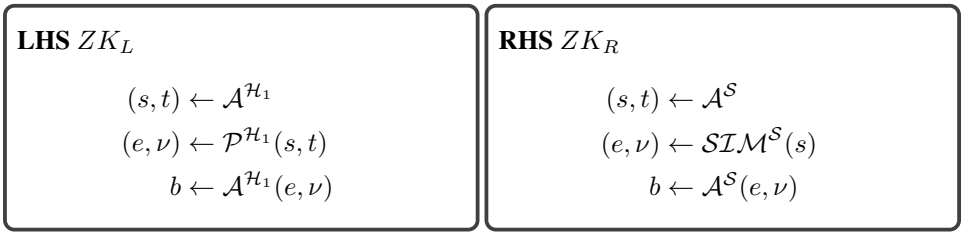
Non-interactive Zero Knowledge An interactive proof that is SHVZK becomes Non-interactive Zero Knowledge (NIZK) when the Fiat-Shamir Transformation is applied to make the proof non-interactive.



Let a left game **LHS** be as above. An adversary $\mathcal{A}$ have access to a random oracle $\mathcal{H}_1$ and outputs a statement s and a witness t. The prover P (with access to the same random oracle) follows the protocol as before, and then sends the proof (e, ν) to $\mathcal{A}$. The adversary then tries to distinguish between the real proof and the simulation.



Let a right game **RHS** be as above, where $Sim^{I}_{eqdl}(aux; g, \bar{g}; x, \bar{x}, e)$ just samples ν at random, $\nu \xleftarrow{\$} \mathbb{Z}_q$, and the simulated hash function $\mathcal{S}$ just samples e at random, $e \xleftarrow{\$} dH_1$. An adversary $\mathcal{A}$ have access to the simulator $\mathcal{S}$ and outputs a statement s and a witness t. The simulator $\mathcal{SIM}$ (with access to $\mathcal{S}$) uses $\mathcal{S}$ to get e, $Sim^{I}_{eqdl}(aux; g, \bar{g}; x, \bar{x}, e)$ to get ν and then sends the proof (e, ν) to $\mathcal{A}$. The adversary then tries to distinguish between the real proof and the simulation.



The advantage of the adversary is

$$\left|\Pr[\mathcal{A}^{\mathcal{H}_1}(e, \nu) = 1 \mid (e, \nu) \xleftarrow{\$} \mathcal{P}^{\mathcal{H}_1}] - \Pr[\mathcal{A}^{\mathcal{S}}(e, \nu) = 1 \mid (e, \nu) \xleftarrow{\$} \mathcal{SIM}^{\mathcal{S}}]\right|.$$

Soundness Probability bounds for the adversary being able to forge proofs when the relation does not hold are given in the following propositions.

Lemma 2.10.1. *Let G be a group of prime order q, and let g and $\bar{g}$ be generators of G. Suppose t and $\Delta, \Delta \neq 0$ are integers, and $g, \bar{g}, x, \bar{x}, \alpha_1, \alpha_2$ are group elements such that $x = g^t$ and $\bar{x} = \bar{g}^{u+\Delta}$. Let e be an integer choosen uniformly at random from $\{1, ..., 2^t au\}$.*

Then the probability that there exists an integer ν such that

$$\alpha_1 x^{-e} = g^{\nu} \text{ and } \alpha_2 \bar{x}^{-e} = \bar{g}^{\nu}$$

is at most $1/2^{\tau}$.

Proof. Let $\alpha_1 = g^u$ and $\alpha_2 = \bar{g}^{u+\delta}$ for δ an integer. Then the relations will be

$$u - et \equiv \nu \bmod q \text{ and } u + \delta - et - e\Delta \equiv \nu \bmod q.$$

For this to be satisfied, we must have that

$$\delta - e\Delta \equiv \nu \bmod q.$$

Both δ and Δ are fixed before e is sampled, so the probability of this event is at most $1/2^{\tau}$. ■

Theorem 2.10.2. *For any algorithm $\mathcal{A}$ that makes at most η queries to the random oracle H_1 and outputs a proof $\pi_{\mathcal{E}}$ and statement s, where the relation $\log_g x = \log_{\bar{g}} \bar{x}$ does not hold, then*

$$\mathcal{V}^{I}_{eqdl}(s, \pi_{\mathcal{E}}) = 1$$

with probability at most $2\eta/2^{\tau}$.

Proof. If the algorithm has not already queried H_1 at the relevant point, the proof verifies with probability $1/2^{\tau}$.

By Lemma 2.10.1, the adversary has a probability of at most $1/2^{\tau}$ of forging a proof every time she queries H_1.

We now have at most $\eta+1$ event, each with probability at most $1/2^{\tau}$, and the probability that at least one of them happen can be bounded as

$$\begin{aligned}
1 - \Big(1 - \frac{1}{2^{\tau}}\Big)^{\eta+1} &= 1 - \sum_{i=0}^{\eta+1} \binom{\eta+1}{i} (-1)^i \Big(\frac{1}{2^{\tau}}\Big)^i \\
&\leq (\eta+1)\frac{1}{2^{\tau}} + \sum_{i=2}^{\eta+1} (\eta+1)^i \Big(\frac{1}{2^{\tau}}\Big)^i \\
&\overset{\eta+1<2^{\tau/2}}{\leq} (\eta+1)\frac{1}{2^{\tau}} + \sum_{i=2}^{\eta+1} \frac{1}{2^{\tau}} \\
&\leq 2\eta\frac{1}{2^{\tau}} \qquad (2.1)
\end{aligned}$$

■

Proof of correct computation II

We need to prove that we have raised a number of group elements to the same power. The proof is $\pi_{\mathcal{T}I}$ with the prover's and the verifier's algorithms as follows:

$$\begin{aligned}
\pi_{\mathcal{T}I} &\leftarrow \mathcal{P}^{II}_{eqdl}(aux, g, \gamma, s; w_1, ..., w_k, \check{w}_1, ..., \check{w}_k) \\
1/0 &\leftarrow \mathcal{V}^{II}_{eqdl}(aux, g, \gamma; w_1, ..., w_k, \check{w}_1, ..., \check{w}_k; \pi_{\mathcal{T}I}).
\end{aligned}$$

The public input which both the prover and the verifier gets, is some auxillary information aux, a generator g, a commitment γ, and the group elements $w_1, ..., w_k, \breve{w}_1, ..., \breve{w}_k$.The prover's private input is the integer s such that $\gamma = g^s$ and $w_i^s = \breve{w}_i,\ i = 1, ..., k$.

Completeness is also required here.

Instantiation The verifier chooses random $e_1, ..., e_k$ from $\{1, ..., 2^\tau\}$ and sends $\vec{e} = (e_1, ..., e_k)$ to the prover.

The prover computes $x = \Pi_{i=1}^k w_i^{e_i}$, chooses u at random from $\mathbb{Z}_q$, computes $\alpha_1 = g^u, \alpha_2 = x^u$ and send (α_1, α_2) to the verifier.

The verifier chooses random e from $\mathbb{Z}_q$ and send it to the prover.

The prover computes $\nu \leftarrow u - se \bmod q$, and sends ν to the verifier.

The verifier computes $x = \Pi_{i=1}^k w_i^{e_i}$ and $\breve{x} = \Pi_{i=1}^k \breve{w}_i^{e_i}$ and accepts iff

$$\alpha_1 = g^\nu \gamma^e \text{ and } \alpha_2 = x^\nu \breve{x}^e.$$

Now, apply the Fiat-Shamir transformation to get the proof non-interactive. Let a hash functions $H_1 : \{0,1\}^* \times G^{2k+2} \to \{1, ..., 2^\tau\}^k$ and $H_2 : \{0,1\}^* \times G^{2k+2} \to \{1, ..., 2^\tau\}$ be evaluated as

$$\begin{aligned} \vec{e} &\leftarrow H_1(aux, g, \gamma, \vec{w}, \vec{\breve{w}}) \\ e &\leftarrow H_2(aux, g, \gamma, \vec{w}, \vec{\breve{w}}, \alpha_1, \alpha_2). \end{aligned}$$

The proof is $\pi_{\mathcal{TI}} = (e, \nu)$ and is accepted iff

$$e = H_2(aux, g, \gamma, \vec{w}, \vec{\breve{w}}, g^\nu \gamma^e, x^\nu \breve{x}^e).$$

Security analysis As before, zero knowledge and soundness is required.

The protocol is SHVZK because there exists a simulator that for a given e, samples ν at random and outputs it, $\nu \leftarrow Sim_{eqdl}^{II}(aux, g, \gamma, \vec{w}, \vec{\breve{w}}, \vec{e}, e)$. Then it is clear that $\alpha_1 = g^\nu \gamma^e$ and $\alpha_2 = x^\nu \breve{x}^e$ with x and $\breve{x}$ as above have the same distribution as in the real protocol.

Applying the Fiat-Shamir transformation then yields a non-interactive zero knowledge proof. Generate this proof by first sampling an e at random, query H_1 to get $\vec{e}$, use Sim_{eqdl}^{II} to get ν, and then reprogram the H_2 oracle appropriately.

This proof is sound in the Random Oracle Model and a soundness bound is given below.

Lemma 2.10.3. *Let V be any proper subspace of $\mathbb{F}_q^k$, and let S be a subset of $\mathbb{F}_q$ with 2^τ elements. Sample $e_1, ..., e_k$ independently and uniformly at random from S. Then the probability that $\vec{e}$ lies inside V is at most $1/2^\tau$.*

Proof. The proof is given in [ste13]. ■

Lemma 2.10.4. *Let G be a group of prime order q and g a generator of it. Let S be a subset of $\mathbb{Z}_q$ with 2^τ elements. Suppose $s, \Delta_1, ..., \Delta_k$ are integers such that $s \not\equiv 0 \bmod q$ and $\Delta_j = 0$ for at least one j. Let $w_1, ..., w_k, \breve{w}_1, ..., \breve{w}_k$ be group elements such that $\breve{w}_i = w_i^s g^{\Delta_i}$, $i = 1, ..., k$.*

If $\Delta_i \not\equiv 0 \bmod q$ for any i, and $e_1, ..., e_k$ are integers chosen uniformly at random from a set with 2^τ elements, then

$$\Pi_{i=1}^{k} \check{w}_i^{e_i} = (\Pi_{i=1}^{k} w_i^{e_i})^s \tag{2.2}$$

holds with probability at most $1/2^\tau$.

Proof. The equation $\Sigma_{i=1}^{k} \Delta_i e_i \equiv 0 \bmod q$ defines a proper subspace of $\mathbb{F}_q^k$. If equation (2.2) holds, then $\Pi_{i=1}^{k} g^{\Delta_i e_i} = 1$, or $\Sigma_{i_1}^{k} \equiv 0 \bmod q$. That is, (2.2) holds only if $\vec{e}$ considered as an $\mathbb{F}_q$-vector falls inside a proper subspace of $\mathbb{F}_q^k$. The claim then follows by Lemma 2.10.3. ■

The last lemma needed for soundness is similar to the one above, 2.10.1, with the proof being similar as well.

Theorem 2.10.5. *For any algorithm that makes at most η queries overall to the random oracles H_1 and H_2, outputs a proof $\pi_{\mathcal{T}I} = (e, \nu)$, a bit string aux, an integer s, and group elements $g, \gamma, w_1, ..., w_k, \check{w}_1, ..., \check{w}_k$ such that $w_i^s \neq \check{w}_i$ for some i, then*

$$\mathcal{V}_{eqdl}^{II}(aux, g, \gamma, \vec{w}, \vec{\check{w}}; \pi_{\mathcal{T}I}) = 1$$

holds with probability at most $2\eta/2^\tau$.

Proof. If the algorithm has not already queried both H_1 and H_2 at the relevant points, the proof verifies with probability $1/2^\tau$.

By Lemma 2.10.4, the adversary has a probability of at most $1/2^\tau$ of forging a proof every time she queries H_1.

By Lemma 2.10.1, the adversary has a probability of at most $1/2^\tau$ of forging a proof for some input by which she cannot already create a forged proof for, every time she queries H_2.

We now have at most $\eta+1$ event, each with probability at most $1/2^\tau$, and the probability that at least one of them happen can be bounded similar as in equation (2.1). ■

Proof of correct computation III

We need to prove that a single group element has been raised to correct, distinct powers. The proof is $\pi_{\mathcal{T}II}$ with the prover's and the verifier's algorithms as follows:

$$\pi_{\mathcal{T}II} \leftarrow \mathcal{P}_{eqdl}^{III}(aux, g, \check{x}, y_{21}, ..., y_{2k}, a_{21}, ..., a_{2k}; \hat{w}_1, ..., \hat{w}_k)$$
$$1/0 \leftarrow \mathcal{V}_{eqdl}^{III}(aux, g, \check{x}; y_{21}, ..., y_{2k}, \hat{w}_1, ..., \hat{w}_k; \pi_{\mathcal{T}II}).$$

Both the prover and the verifier receive the public input consisting of some auxilliary information aux, a generator g of G, the commitments $y_{2i}, ..., y_{2k}$, the base $\check{x}$ and the powers $\hat{w}_1, ..., \hat{w}_k$. The prover in addition gets the private input the integers $a_{21}, ..., a_{2k}$.

Completeness is also required here.

Instantiation The verifier chooses random $e_1, ..., e_k$ from $\{1, ..., 2^\tau\}$ and sends $\vec{e} = (e_1, ..., e_k)$ to the prover.

The prover chooses u at random from $\mathbb{Z}_q$, computes $\alpha_1 = g^u, \alpha_2 = \breve{x}^u$ and send (α_1, α_2) to the verifier.

The verifier chooses random e from $\mathbb{Z}_q$ and send it to the prover.

The prover computes $\nu \leftarrow u - e\Sigma_{i=1}^k e_i a_{2i} \bmod q$, and sends ν to the verifier.

The verifier computes $y_2 = \Pi_{i=1}^k y_{2i}^{e_i}$ and $\hat{w} = \Pi_{i=1}^k \hat{w}_i^{e_i}$, and accepts iff

$$\alpha_1 = g^\nu y_2^e \text{ and } \alpha_2 = \breve{x}^\nu \hat{w}^e.$$

Now, apply the Fiat-Shamir transformation to get the proof non-interactive. Let a hash functions $H_1 : \{0,1\}^* \times G^{2k+2} \to \{1, ..., 2^\tau\}^k$ and $H_2 : \{0,1\}^* \times G^{2k+2} \to \{1, ..., 2^\tau\}$ be evaluated as

$$\vec{e} \leftarrow H_1(aux, g, \breve{x}, \vec{y_2}, \vec{\hat{w}})$$
$$e \leftarrow H_2(aux, g, \breve{x}, \vec{y_2}, \vec{\hat{w}}, \alpha_1, \alpha_2).$$

The proof is $\pi_{\mathcal{T}II} = (e, \nu)$ and is accepted iff

$$e = H_2(aux, g, \gamma, \vec{w}, \vec{\hat{w}}, g^\nu y_2^e, \breve{x}^\nu \hat{w}^e).$$

Security analysis As before, zero knowledge and soundness is required.

The protocol is SHVZK because there exists a simulator that for a given e, samples ν at random and outputs it, $\nu \leftarrow Sim_{eqdl}^{III}(aux, g, \breve{x}, \vec{y_2}, \vec{\hat{w}}, \vec{e}, e)$. Then it is clear that $\alpha_1 = g^\nu y_2^e$ and $\alpha_2 = \breve{x}^\nu \hat{w}^e$ with y_2 and $\hat{w}$ as above have the same distribution as in the real protocol.

Applying the Fiat-Shamir transformation then yields a non-interactive zero knowledge proof. Generate this proof by first sampling an e at random, query H_1 to get $\vec{e}$, use Sim_{eqdl}^{III} to get ν, and then reprogram the H_2 oracle appropriately.

This proof is sound in the random oracle model and a soundness bound is given below.

Lemma 2.10.6. *Let G be a group of prime order q and g a generator of it. Suppose $a_{21,...,a_{2k}}, \Delta_1, ..., \Delta_k$ are integers and $\breve{x}, y_{21}, ..., y_{2k}, \hat{w}_1, ..., \hat{w}_k$ group elements such that $y_{2i} = g^{a_{2i}}$ and $\hat{w}_i = \breve{x}^{a_{2i}} g^{\Delta_i}$, $i = 1, ..., k$.*

If $\Delta_i \not\equiv 0 \bmod q$ for any i, and $e_1, ..., e_k$ are integers chosen uniformly at random from a set with 2^τ elements, then the probability that there exists an integer a such that

$$\Pi_{i=1}^k y_{2i}^{e_i} = g^a \text{ and } \Pi_{i=1}^k \hat{w}_i^{e_i} = \breve{x}^a \tag{2.3}$$

holds with probability at most $1/2^\tau$.

Proof. If the left part of (2.3) holds, then $\Pi_{i=1}^k g^{\Delta_i e_i} = 1$ or $\Sigma_{i=1}^k \Delta_i e_i \equiv 0 \bmod q$. So (2.3) only holds if $\vec{e}$ considered as an $\mathbb{F}_q$-vector falls inside the previously defines proper subspace of $\mathbb{F}_q^k$. The claim then follows by Lemma 2.10.3. ■

The last lemma needed for soundness is similar to the one above, Lemma 2.10.1, with the proof being similar as well.

Theorem 2.10.7. *For any algorithm that makes at most η queries overall to the random oracles H_1 and H_2, outputs a proof $\pi_{\mathcal{TII}} = (e, \nu)$, a bit string aux, integers $a_{21}, ..., a_{2k}$, and group elements $g, \check{x}, y_{21}, ..., y_{2k}, \hat{w}_1, ..., \hat{w}_k$ such that $y_{2i} = g^{a_{2i}}$ for all i, but $\check{x}^{a_{2i}} \neq \hat{w}_i$ for some i, then*

$$\mathcal{V}^{III}_{eqdl}(aux, g, \check{x}, \vec{y_2}, \vec{\hat{w}}; \pi_{\mathcal{TII}}) = 1$$

holds with probability at most $2\eta/2^\tau$.

Proof. If the algorithm has not already queried both H_1 and H_2 at the relevant points, the proof verifies with probability $1/2^\tau$.

By Lemma 2.10.6, the adversary has a probability of at most $1/2^\tau$ of forging a proof every time she queries H_1.

By Lemma 2.10.1, the adversary has a probability of at most $1/2^\tau$ of forging a proof for some input by which she cannot already create a forged proof for, every time she queries H_2.

We now have at most $\eta+1$ event, each with probability at most $1/2^\tau$, and the probability that at least one of them happen can be bounded similar as in equation (2.1). ■

Chapter 3

Formal verification

Formal verification is proving the correctness of algorithms with respect to formal specifications. Tools for formal verification may be used to prove (or disprove) the correctness of cryptographic protocols. The pen and paper proofs often uses the provable security approach, in which mathematical problems are tried reduced to attacks on the cryptographic construction. These proofs are often long and complicated. Tools where one may formalize systems and write proofs, and interactively verify steps on the way, may be used to increase confidence in such reductionist proofs.

3.1 EasyCrypt

EASYCRYPT is a tool for reasoning about probabilistic relational properties and the security of cryptographic constructions with adversial code. Is is used to construct and verify game-based cryptographic proofs. Four logics are used: Hoare logic, probabilistic Hoare logic, relational probabilistic Hoare logic, and ambient logic. SMT-solvers are able to automatically prove certain simple ambient logic goals.

EASYCRYPT has been used to provide machine-checked proof of privacy-related properties (including ballot privacy) for an electronic voting protocol in the computational model [War17] [cat17].

3.2 SMT-solvers

SMT (satisfiability modulo theories) are decision problems for logical formulas. SMT-solvers are tools to automatically solve SMT problems. It is useful for verification and program correctness proving. In interactive theorem proving and computer-aided program verification, SMT solvers may be used to automatically verify (or reject) proof steps. There are several techniques for this, but one is to translate assertions, pre-, post-, and possibly loop-conditions or conditionals into SMT formulas so that it can be decided if all properties

hold. SMT solvers will take as input a first-order logic formula F over a ground theory T and return if it is satisfiable or not.

EASYCRYPT uses the the SMT solvers Alt-Ergo, Z3 and E by default, but more are available if wanted.

Alt-Ergo Alt-Ergo is an SMT solver dedicated to prove mathematical formulas in program verification. It provides support for theories including empty theory, linear integer and rational arithmetic, non-linear arithmetic, polymorphic arrays, enumerated datatypes, bitvectors, and quantifiers.

E E is a theorem prover for full first-order logic with equality. It accepts a problem specification (typically consisting of a number of first-order clauses or formulas) and a conjecture, and will then try to find a formal proof for the conjecture, assuming the axioms. If a proof is found, proof steps that can be individually verified is provided.

Z3 Z3 is a SMT solver with support for theories including empty theory, linear arithmetic, nonlinear arithmetic, bitvectors, arrays, datatypes, quantifiers, and strings. Z3 is commonly used in program verification.

3.3 Problems

During the development of the modelling of the system and writing of the proofs, some problems occurred. A couple of bugs were found in the SMT solvers Alt-Ergo and E. We were able to prove false when a real divided with a real were in context. This resulted in us being able to prove that the order of a prime-order group, or that an arbitrary prime, was 1. The problematic SMT solvers were blocked. This lead to some of the proofs not working and they needed to be rewritten.

Chapter 4

Simplified protocol

In [ste10] from 2010, a simplified protocol is specified and analyzed.

The players in the protocol are the voter V, the voter's computer P, the ballot box B, the return code generator R, and the decryption service D. The auditor A is not part of the simplified protocol.

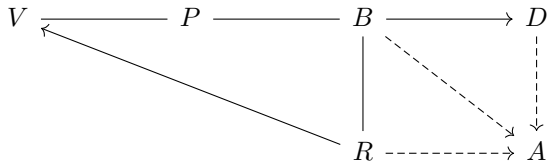


Figure 4.1: The players and the communication canals. Figure 1 in [ste10].

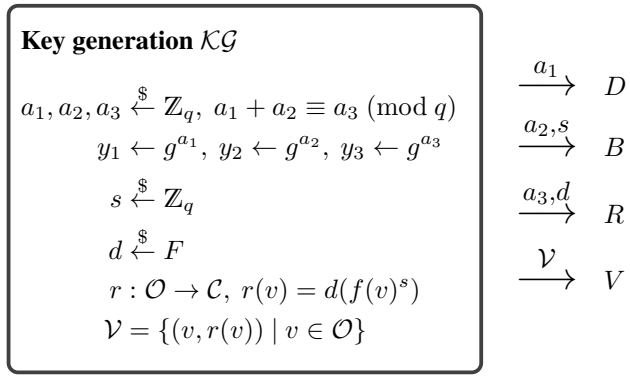
The infrastructure players are B, R, D and A. The players communicate via secure, authenticated channels. In the simplified protocol, the ballot box knows which voter is communicating with which computer. The players in the protocol and their communication is illustrated in Figure 4.1.

The voter chooses as her ballot a sequence of options $(v_1, ..., v_k)$ from a set of options $\mathcal{O} = \{1, 2, ...\}$. The computer then pads this sequence with zeros for $k < i \leq k_{max}$, encrypts it with the election encryption key, and then submits the encrypted ballot to the ballot box. Then the ballot box and the return code generator computes return codes from a set $\mathcal{C}$ for each voter that are sent directly to the voter. The voter can check if the return code she received matches the options she selected. If it matches the voter accepts, and if not, something went wrong.

Prerequisites The system uses a finite cyclic group G of prime order q with generator g, pseudo-random function family $F : G \to \mathcal{C}$, an injective encoding function

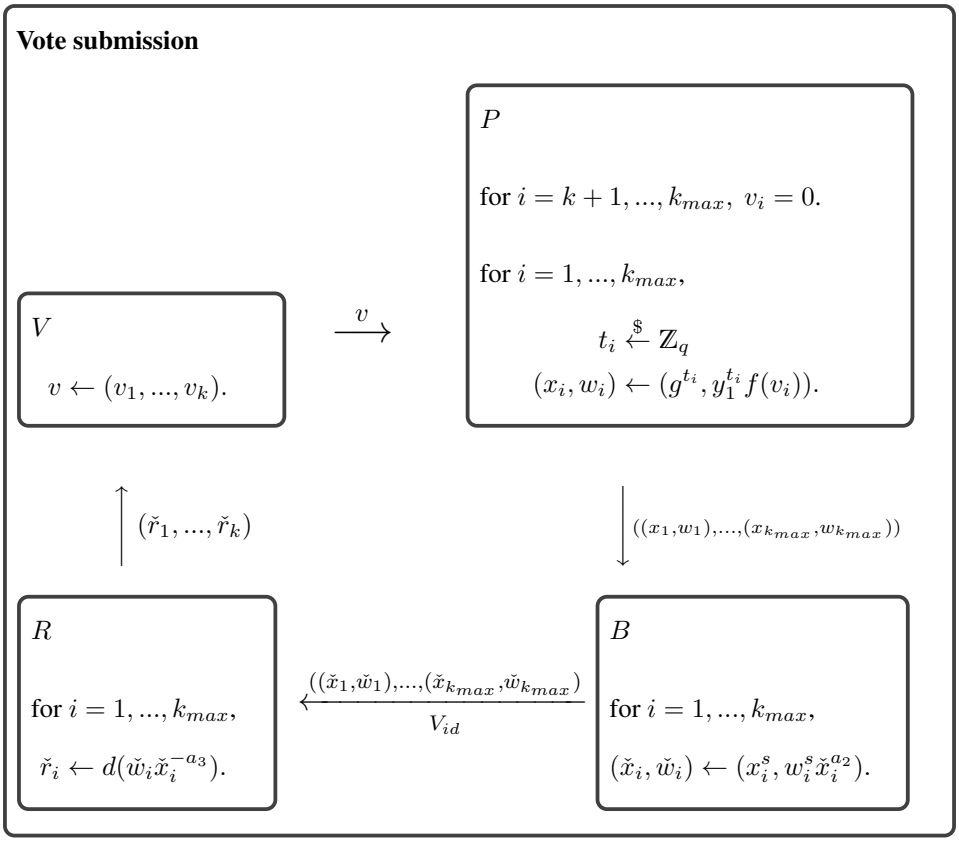
$f : \mathcal{O} \to G$, $f(0) = g_{id}$, where g_{id} is the identity element in G.

Key generation The key generation is assumed done by a trusted third party. It is done before the election.



The key generation algorithm generates the private keys a_1, a_2, a_3 and the public keys y_1, y_2, y_3, and send a_1 to D, a_2 to B and a_3 to R. In addition, an integer s and a function d is sampled for every voter, such that every voter V is given a set with pairs $(v, d(f(v)^s))$ for each option v.

Vote submission The vote submission has four players. The voter V wants to submit the ballot $v = (v_1, ..., v_k)$ with k options as her vote.



The vote submission goes as follows. The voter V sends her ballot $(v_1, ..., v_k)$ to P, which pads it with zeros from k to k_{max}, encrypts it and sends it to B. The ballot box then computes using the integer s for the voter and sends it, in addition to the voter's identity V_{id}, to R. Now R generates return codes which it sends to V. Lastly, the voter verifies that pair of options and return codes are in the set of return codes which she received before the election. The voter is able to submit several ballots. If a new ballot is submitted the previous are superseded by it.

In Figure 4.2 the protocol for submission of one option and generation of one return code is illustrated.

Counting The voter V has verified that every pair $(v_i, \check{r}_i)$ is in the set of return codes $\mathcal{V}$ received by $\mathcal{KG}$ before the election, and if so considered the ballot cast.

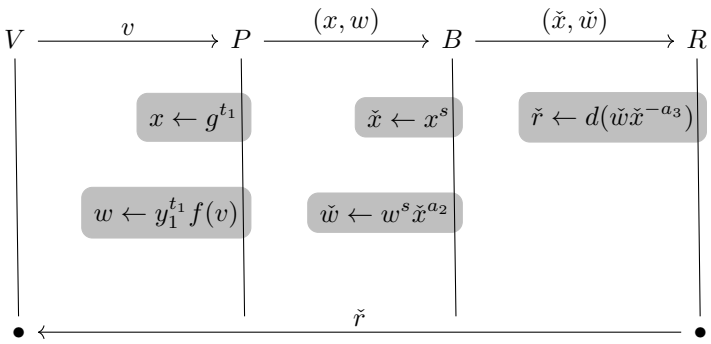


Figure 4.2: Vote submission. Figure 2 in [ste10].

Counting

D

for $i = 1, ..., k_{max}$,

$$\mu_i \leftarrow w_i x_i^{-a_1}.$$

Output the resulting ballots in random order.

When the ballot box B closes, superceded ballots are discarded. Then B sends the remaining encrypted ballots $((\check{x}_1, \check{w}_1), ..., (\check{x}_{k_{max}}, \check{w}_{k_{max}}))$ to the decryption service D, in random order. In D all the ciphertexts are decrypted, and the resulting ballots output in random order.

4.1 Completeness

The protocol is complete if, when all players are honest, the submitted ballots are correctly decrypted and the return codes match the ones in the set the voter received before the ballot submission.

We have completeness in the return code received by the voter if $(v, \check{r})$ is in $\mathcal{V}$, that is, that $\check{r} = r(v)$. Completeness follows from computing

$$\check{w}\check{x}^{-a_3} = (w^s \check{x}^{a_2})\check{x}^{-a_3} = w^s \check{x}^{-a_1} = w^s (x^s)^{-a_1} = (wx^{-a_1})^s = f(v)^s.$$

4.2 Security

This section will argue for the security of the simplified protocol. The corruption models considered are:

(a) The voter and its computer are corrupted.

(b) The voter's computer is corrupt.

(c) One of the tree infrastructure players is passively corrupt (or honest-but-curious).

The focus will be on (c), and especially an honest-but-curious corruption in the ballot box and return code generator. Honest-but-curious means that the adversary follows the protocol, but will try to deduce information about the voter's ballots. The two following properties are argued for:

1. The voter will most likely notice if a corrupt computer modifies a ballot,
2. No honest-but-curious infrastructure player will learn any non-trivial information about the ballots,

with a focus on the latter one.

4.2.1 (a) Voter and computer

There is assumed to be authenticated channels, and the ballot box can therefore ensure that only one ballot is counted per voter. If the ballot's ciphertext is malformed, it will invalidate the ballot.

4.2.2 (b) Computer

Suppose the computer submits $(v'_1, ..., v'_k) \in \mathcal{O}'$ instead of $(v_1, ..., v_k) \in \mathcal{O}$, where $\mathcal{O}' \supseteq \mathcal{O}$. Since f is an injection, exponentiation in G just is a permutation, and d is a random looking function, the composition of f, a permutation and d will look like a random function from $\mathcal{O}'$ to $\mathcal{C}$.

Any function from $\mathcal{O}'$ to $\mathcal{C}$ defines a partition of $\mathcal{O}'$, which again defines an equivalence relation. The uniform distribution on the set of functions from $\mathcal{O}'$ to $\mathcal{V}$ therefore induces a probability distribution on the set of equivalence relations on $\mathcal{O}'$. Let $\sim$ be an equivalence relation and extend it on $\mathcal{O}'$ such that $(v_1, ..., v_k) \sim (v'_1, ..., v'_k)$ iff $k = k'$ and $v_i \sim v'_i,\ i = 1, ..., k$. The voter will accept the manipulation iff $(v'_1, ..., v'_k) \sim (v_1, ..., v_k)$. Assuming the set $\mathcal{C}$ is sufficiently large, the probability of this event will be small.

4.2.3 (c) Infrastructure players

We consider the three infrastructure players B, R and D. All of them are honest-but-curious, so we will only need to simulate the input they normally would see, and will not need to model interaction with other players in the system. First B is considered, then R, and finally, D.

The ballot box We are given a finite cyclic group G with generator g and prime order q. Let $\mathbb{Z}_q = \{0, ..., q-1\}$ be the set where keys and exponents are sampled from.

Suppose we have an honest-but-curious ballot box B^* that after the election is over looks at the ciphertexts and outputs some non-trivial information about the ballots submitted. This B^*-adversary, along with a DDH-distinguisher, will be modelled in EASYCRYPT. They will be used in games that simulates the view of B^*.

We want to show that a simulated input from **DDH0** (defined in §2.5) simulates the ballot box's input perfectly. To do this, we will play a IND-CPA game (indistinguishability under chosen-plaintext attack) where we use the key generation algorithm $\mathcal{KG}$ and the computer P.

Key generation $\mathcal{KG}$

Generate a_1, a_2, a_3 such that $a_1 + a_2 \equiv a_3 \pmod{q}$.

Compute $y_i = g^{a_i}$.

Output $(a_1, a_2, a_3, y_1, y_2, y_3)$.

Computer P

Receive public key y_1 and an option v.

$$t \xleftarrow{\$} \{0, ..., q-1\}$$

Encrypt v and output $(g^t, y_1^t f(v))$.

IND-CPA

$$\begin{aligned}(a_1, a_2, a_3, y_1, y_2, y_3) &\leftarrow \mathcal{KG} \\ (v_0, v_1) &\leftarrow B^*(a_2) \\ b &\xleftarrow{\$} \{0,1\} \\ (x, w) &\leftarrow P(y_1, v_b) \\ b' &\leftarrow B^*(x, w)\end{aligned}$$

B^* wins if $b' = b$.

```
module KG1 = {
  proc kg() = {
    var a1, a2, a3, y1, ¦
    y2, y3;
    a1 <$ dt;
    a2 <$ dt;
    a3 <- a1 + a2;
    y1 <- g^a1;
    y2 <- g^a2;
    y3 <- g^a3;
    return (a1, a2, a3, ¦
    y1, y2, y3);
  }
}.

module P1 = {
  proc enc(y1 : group, v : ¦
    int) = {
    var t, x, w;
    t   <$ dt;
    x   <- g^t;
    w   <- y1^t * (f v);
    return (x, w);
  }
}.

module INDCPA (B' : HbC) = {
  proc main() = {
    var a1, a2, a3, y1, ¦
    y2, y3, m0, m1, x, w, ¦
    b, b';
    (a1, a2, a3, y1, y2, ¦
    y3) <@ KG1.kg();
    (m0, m1) <@ ¦
    B'.choose(a2);
    b        <$ {0,1};
    (x, w)   <@ P1.enc(y1, ¦
    b?m1:m0);
    b'       <@ ¦
    B'.guess(x, w);
    return b' = b;
  }
}.
```

The keys are obtained from $\mathcal{KG}$ and a_2 is sent to B^*. Two chosen options v_0 and v_1 are outputted by B^*. A bit b is sampled at random, and v_b, along with y_1, is sent to P for

encryption. The ciphertext (x, w) obtained from P is given to B^*, and B^* makes a guess b' on which of v_0 and v_1 that were encrypted. If $b' = b$, B^* has won the game.

Adversaries
Two abstracts adversaries. The DDH-adversary `DDHADV` will later be instantiated, while the honest-but-curious ballot box `HbC` will remain abstract.

```
module type DDHADV = {
  proc guess(gx gy gz : ¦
    group) : bool
}.

module type HbC = {
  proc choose(gx : t) : ¦
    int * int
  proc guess(gy gzm : ¦
    group) : bool
}.
```

The Decision Diffie-Hellman problem from §2.5 is used in the two games **DDH0** and **DDH1** below.

DDH0

$$a_1 \xleftarrow{\$} \mathbb{Z}_q$$
$$y_1 \leftarrow g^{a_1}$$
$$u_1 \xleftarrow{\$} G$$
$$u_2 \leftarrow v^{a_1}$$

Send DDH-tuple (y_1, u_1, u_2) to the simulator and receive b. B^* has won if $b = 1$.

```
module DDH0 (A : DDHADV) = {
  proc main() = {
    var b, a1, y1, u1, u2;
    a1 <$ dt;
    y1 <- g^a1;
    u1 <$ dG;
    u2 <- u1^a1;
    b <@ A.guess(y1, u1, u2);
    return b;
  }
}.
```

DDH1

$$a_1 \xleftarrow{\$} \mathbb{Z}_q$$
$$y_1 \leftarrow g^{a_1}$$
$$u_1 \xleftarrow{\$} G$$
$$u_2 \xleftarrow{\$} G$$

Send random tuple (y_1, u_1, u_2) to the simulator and receive b. B^* has won if $b = 1$.

```
module DDH1 (A : DDHADV) = {
  proc main() = {
    var b, a1, y1, u1, u2;
    a1 <$ dt;
    y1 <- g^a1;
    u1 <$ dG;
    u2 <$ dG;
    b <@ A.guess(y1, u1, u2);
    return b;
  }
}.
```

In **DDH0** a DDH-tuple is generated and outputted, and in **DDH1** a random tuple is generated and outputted.

We will have a reduction where we, from the B^*-adversary, construct a DDH-adversary as below.

DDH-adversary

On input (y_1, u_1, u_2).

Generate a_2.

Compute $y_3 = y_1 g^{a_2}$.

Send a_2 to B^* and receive cleartext options m_0, m_1.

$$b \xleftarrow{\$} \{0,1\}.$$

$$t, t' \xleftarrow{\$} \{0, ..., q-1\}.$$

Encrypt m_b as $(g^t u_1^{t'}, y_1^t u_2^{t'} f(m_b))$ and send the ciphertext to B^*.

B^* outputs its guess b' and wins if $b' = b$.

```
module DDHAdv (B' : HbC) = {
  var bad : bool
  proc guess(y1, u1, u2) = {
    var a2, y3, m0, m1, t, ¦
    t', b, b';
    a2        <$ dt;
    y3        <- y1*g^a2;
    (m0, m1) <@ ¦
    B'.choose(a2);
    b         <$ {0,1};
    t         <$ dt;
    t'        <$ dt;
    bad       <- (t + ¦
    log(u1) * t') = F.zero ¦
    \/ - t' * log u1 + (t ¦
    + log(u1) * t') = ¦
    F.zero;
    b'        <@ ¦
    B'.guess(g^t*u1^t', ¦
    y1^t*u2^t'*f(b?m1:m0));
    return b' = b;
  }
}.
```

The DDH-adversary above will get an input tuple (y_1, u_1, u_2) either from **DDH0** or **DDH1.** It generates a_2 which it sends to B^*, which responds with two options m_0 and m_1. A bit b is sampled at random, before t, t' are sampled at random from $\mathbb{Z}_q$. The encryption of m_b is sent to B^*. Notice that the encryption here is not the usual $(g^t, y_1^t f(m_b))$, but instead the elements u_1, u_2 from the input tuple are used as $(g^t u_1^{t\prime}, y_1^t u_2^{t'} f(m_b))$.

To show that the simulated input from **DDH1** contains no information about the ballots, we will use a game **Gb.**

Gb

Generate a_1, a_2, a_3 such that $a_1 + a_2 \equiv a_3 \pmod{q}$.

Compute $y_i = g^{a_i}$.

$$
\begin{aligned}
(v_0, v_1) &\leftarrow B^*(a_2) \\
b &\xleftarrow{\$} \{0,1\} \\
t &\xleftarrow{\$} \{0, ..., q-1\} \\
(x, w) &\leftarrow (g^t, y_1^t) \\
b' &\leftarrow B^*(x, w)
\end{aligned}
$$

B^* wins if $b' = b$.

Guess

The ciphertext B^* receive contains no information about the option, so its probability of winning is equal to $1/2$.

```
module Gb (B' : HbC) = {
  var bad : bool
  proc main() = {
    var a1, a2, a3, y1, ¦
    y2, y3, m0, m1, t, t', ¦
    b, b', u1, u2;
    a1        <$ dt;
    a2        <$ dt;
    a3        <- a1 + a2;
    y1        <- g^a1;
    y2        <- g^a2;
    y3        <- g^a3;
    u1        <$ dG;
    u2        <$ dG;
    (m0, m1) <@ ¦
    B'.choose(a2);
    b         <$ {0,1};
    t         <$ dt;
    t'        <$ dt;
    bad       <- t = F.zero ¦
    \/ - t' * log u1 + t = ¦
    F.zero;
    b'        <@ ¦
    B'.guess(g^t, y1^t);
    return b' = b;
  }
}.

lemma Gb_half (B' <: HbC) ¦
    &m :
  Pr[Gb(B').main() @ &m : ¦
    res] = 1%r / 2%r.
```

The keys are generated by **Gb** the usual way before a_2 is sent to B^*. As in **IND-CPA,** B^* outputs two options v_0 and v_1 and a bit b is sampled at random. Now the ciphertext (x, w) is computed as $(x, w) = (g^t, y_1^t)$ with the randomly sampled t. Since this ciphertext contains no information about the ballots, B^*'s probability of winning is like a coin flip.

Now all the games are modelled, so all we need to do to make a conclusion is to express some intermediate lemmas.

The first lemma we will need is one that states that

```
INDCPA(B') ~ DDH0(DDHAdv(B')).
```

DDH-tuple
If the given tuple (y_1, u_1, u_2) from G is a DDH-tuple, the simulation will simulate the ballot box input perfectly.

```
lemma B_DDH0 (B' <: HbC {DDHAdv}) &m :
  Pr[INDCPA(B').main() @ &m : res] = ¦
    Pr[DDH0(DDHAdv(B')).main() @ &m : res].
```

The input to B^* from `INCPA` is as to the ballot box B, $(g^t, y_1^t f(v_b))$, and the input to B^* from `DDH0(DDHAdv)` is $(g^t u_1^{t'}, y_1^t u_2^{t'} f(v_b))$. Since

$$\begin{aligned}(g^t u_1^{t'}, y_1^t u_2^{t'} f(v_b)) &= (g^t u_1^{t'}, (g^{a_1})^t (u_1^{a_1})^{t'} f(v_b)) \\ &= (g^{t+t' \log_g u_1}, g^{a_1(t+t' \log_g u_1)} f(v_b)),\end{aligned}$$

we have that

$$(g^t, y_1^t f(v_b)) \sim (g^t u_1^{t'}, y_1^t u_2^{t'} f(v_b)).$$

In the proof an isomorphism $t \pm t' \log_g u_1$ is made use of when sampling t to get this result.

The second lemma we will need is one that states that (for large q)

```
Gb(B') ~ DDH1(DDHAdv(B')).
```

Here we will need to bound the equivalence by $2/q$.

Random tuple
If the given tuple (y_1, u_1, u_2) from G is a random tuple, the input will contain no information about the ballots.

```
lemma Gb_DDH1 (B' <: HbC {Gb, DDHAdv}) &m :
  `| Pr[Gb(B').main() @ &m : res] - ¦
    Pr[DDH1(DDHAdv(B')).main() @ &m : res] | <=
  Pr[DDH1(DDHAdv(B')).main() @ &m : DDHAdv.bad].

lemma Bad (B' <: HbC {Gb, DDHAdv}) &m :
  Pr[DDH1(DDHAdv(B')).main() @ &m : DDHAdv.bad] <= 2%r / q%r.
```

The simulation with (y_1, u_1, u_2) from **DDH1** will be equivalent to a game which produces an input to B^* that contains no information about the option. The input to B^* in `Gb` is (g^t, y_1^t), and the input to B^* in `DDH1(DDHAdv)` is $(g^t u_1^{t'}, y_1^t u_2^{t'} f(v_b))$ (with both random u_1 and u_2). The input to B^* from `Gb` will therefore be like a coin flip, and B' therefore has a probability $1/2$ of a successful guess.

The bad event `DDHAdv.bad` is that t is sampled to be equal to 0 or $t' \log_g u_1$. The probability of this happening is at most $2/q$. The need for this bound and additional lemma is due to a need of isomorphisms between the samplings and avoiding to divide by 0 (also, if $t = 0$, one input will just be two identity elements of G, (g_{id}, g_{id}), and the other an identity element and $f(v_b)$, $(g_{id}, f(v_b))$). The isomorphisms are:

for t

$$\bullet\ t - t' \log_g u_1 \quad \bullet\ t + t' \log_g u_1$$

and for a_1

$$\bullet \begin{cases} \frac{a_1 t - t' \log_g u_2 - \log_g f(v_b)}{t - t' \log_g u_1}, & t \neq 0, t - t' \log_g u_1 \neq 0 \\ a_1, \text{ otherwise} \end{cases}$$

$$\bullet \begin{cases} \frac{a_1(t - t' \log_g u_1) + t' \log_g u_2 + \log_g f(v_b)}{t}, & t \neq 0, t - t' \log_g u_1 \neq 0 \\ a_1, \text{ otherwise.} \end{cases}$$

After the use of these isomorphisms, we get that

$$(g^t, y_1^t) \sim (g^t u_1^{t'}, y_1^t u_2^{t'} f(v_b)),$$

and a probability bound of $2/q$ by the lemma `Bad`.

By the help of these lemmas we may prove the final conclusion.

Conclusion
If B^* can extract some information about the ballots, we have a distinguisher for the Decision Diffie-Hellman problem.

```
lemma Conclusion (B' <: HbC {Gb, DDHAdv}) &m :
  `| Pr[INDCPA(B').main() @ &m : res] - 1%r/2%r | <=
  `| Pr[DDH0(DDHAdv(B')).main() @ &m : res] -
     Pr[DDH1(DDHAdv(B')).main() @ &m : res] | + 2%r / q%r.
```

The advantage of the DDH-adversary `DDHAdv` is

```
`| Pr[DDH0(DDHAdv(B')).main() @ &m : res]
 - Pr[DDH1(DDHAdv(B')).main() @ &m : res] | .
```

The last term, $2/q$ (which is negligible in q), is due to the bad event `DDHAdv.bad` explained above. The advantage of B^* to distinguish between the real input and the input containing no information about the ballot,

```
`| Pr[INDCPA(B').main() @ &m : res] - 1%r/2%r | ,
```

is therefore close to zero (assuming a large q).

The return code generator (RCG) Let R^* be an honest-but-curious return code generator that after the election is over outputs some non-trivial information about the ballots submitted. This will be modelled in EASYCRYPT as following.

We assume that the family of functions from $\mathcal{O}$ to G given by $v \mapsto f(v)^s$ is a pseudo-random function family. The functions used in the games **PRFr** and **PRFi** in §2.7 in EASYCRYPT looks like:

```
module PRFr = {
  proc keygen(): K = {
    var k;

    k <$ dK;
    return k;
  }

  proc f(k:K,x:D): R = {
    return F k x; }
}.
```

```
module PRFi = {
  var m:(D,R) fmap

  proc init(): unit = { m
    = empty; }

  proc f (x:D): R = {
    if (x \notin m) m.[x]
    = $dR;
    return (oget m.[x]);
  }
}.
```

Here the left game `PRFr` samples a key at random and uses a real function from the family, while the right game `PRFi` is the ideal one. The right game initializes a key-value-correspondence, and then has as its function a model of a random function. The right module's procedure `f` takes as input a key `k` from its keyspace `K` $= \mathbb{Z}_q$ and an argument `x` from its domain `D` $= \mathcal{O}$, and has range `R` $= G$. The variable `m : (D, R) fmap` in `PRFi` is the set of random finite maps from the domain to the range.

Given $f, \rho : \mathcal{O} \to G$ and $j,\ 1 \leq j \leq N$, let

$$\rho_l : \mathcal{O} \to G,\ 1 \leq l < j, \quad \rho_l = \rho,\ l = j, \quad \rho_l : v \mapsto f(v)^{s_l},\ j < l \leq N.$$

The given functions f and ρ are modelled in EASYCRYPT as operators that have their associated properties stated as axioms. The integers j and N will also be operators with axioms that they are integers less than or equal to one. The set of functions from the family $v \mapsto f(v)^s$ is modelled as the operator `F`.

Given

$f : \mathcal{O} \to G,\ f(0) = g_{id},$
$\rho : \mathcal{O} \to G,$
$j,\ 1 \leq j \leq N.$

For $V_1, ..., V_{j-1}$,

$\rho_l \xleftarrow{\$} \{r \mid r : \mathcal{O} \to G\},\ 1 \leq l < j.$

For V_j,

$\rho_l \leftarrow \rho,\ l = j.$

For $V_{j+1}, ..., V_N$,

$s_l \xleftarrow{\$} \mathbb{Z}_q$
$\rho_l \leftarrow f(v)^{s_l},\ j < l \leq N.$

```
op f : {O -> group | f(0) ¦
    = g1 && injective f} ¦
    as encFunc.
op rho : O -> group.
op N : {int | 1 <= N} as N_.
op j : {int | 1 <= j <= N} ¦
    as j_.

op F = fun (s : t) (v : ¦
    O), f(v)^s.

module PRFb : H.Orclb = {
  proc leaks () : unit = { }
  proc orclL = ¦
    PRF_Wrap(PRFr).f
  proc orclR = PRFi.f
}.

module type Orcl = {
  proc orcl(m : O) : group
}.

module HybOrcl (Ob : ¦
    Orclb, O : Orcl) = {
  var l, l0 : int
  proc orcl(m : O): group ¦
    = {
    var r : group;
    if   (l0 < l) r <@ ¦
    Ob.orclL(m);
    elif (l0 = l) r <@ ¦
    O.orcl(m);
    else          r <@ ¦
    Ob.orclR(m);
    l <- l + 1;
    return r;
  }
}.
```

The encoding function `f` is as before. As before in f, the encoding of zero is the identity element in G, here given by `g1`, and it is injective. Let `rho` be the given function ρ such that `rho : O -> G`, and `F` be the PRF operator that is used in `PRFr`. Then the functions from the PRF family will be $v \mapsto f(v)^s$ as required.

The hybrid oracle `HybOrcl` is used in the above game as follows. For the given j, the PRF family `F` in `orclL = PRF_Wrap(PRFr).f` is used to get the function ρ_l if $j < l$. Else, if $j = l$, the given ρ is returned (here modelled as `Orcl.orcl`). Else, the procedure

`orclR = PRFi.f` checks if the option `v` is already in the domain, and if not, samples a value from the range and returns it.

For voter V_l with ballot $(v_1, ..., v_{k_{max}})$ and function ρ_l, we compute

$$(\check{x}_i, \check{w}_i) = (g^{t'_i}, y_3^{t'_i} \rho_l(v_i)).$$

If $j = 1$ and the given ρ comes from the function family, this simulates R perfectly. Whereas, if $j = N$ and the given ρ is random, R^* can extract no non-trivial information about the ballots. After a hybrid argument, we see that if R^* can extract some information about the ballots, we have a distinguisher for the function family.

This will be modelled in EASYCRYPT as the following games and lemmas similar to in §4.2.3.

Key generation $\mathcal{KG}$ is the same as before, except this time also s is outputted.

Ballot box B

On input $(a_2, s, (x, w))$,

$$(\check{x}, \check{w}) \leftarrow (x^s, w^s \check{x}^{a_2}).$$

Output $(\check{x}, \check{w})$.

IND-CPA

$$\begin{aligned}(a_1, a_2, a_3, y_1, y_2, y_3, s) &\leftarrow \mathcal{KG}\\ (v_0, v_1) &\leftarrow R^*(a_3)\\ b &\xleftarrow{\$} \{0,1\}\\ (x, w) &\leftarrow P(y_1, v_b)\\ (\check{x}, \check{w}) &\leftarrow B(a_2, s, (x, w))\\ b' &\leftarrow R^*(\check{x}, \check{w})\end{aligned}$$

R^* wins if $b' = b$.

```
module B1 = {
  proc enc(a2 s : t, x_w : ¦
    (group * group)) : ¦
    (group * group) = {
    var x', w' : group;
    x'    <- x_w.`1^s;
    w'    <- x_w.`2^s * ¦
    x'^a2;
    return (x', w');
  }
}.

module INDCPA' (R' : HbC) ¦
    = {
  var bad : bool
  proc main() = {
    var a1, a2, a3, y1, ¦
    y2, y3, s, m0, m1, x, ¦
    w, x', w', b, b';
    (a1, a2, a3, y1, y2, ¦
    y3, s) <@ KG1.kg();
    (m0, m1) <@ ¦
    R'.choose(a3);
    b        <$ {0,1};
    (x, w)   <@ P1.enc(y1, ¦
    b?m1:m0);
    (x', w') <@ B1.enc(a2, ¦
    s, (x, w));
    bad      <- s = F.zero;
    b'       <@ ¦
    R'.guess(x', w');
    return b' = b;
  }
}.
```

The **IND-CPA** game is similar to the one for B^*, but this time both P and B are used to obtain the ciphertext $(\check{x}, \check{w})$ which is sent to R^*.

Gb

Generate a_1, a_2, a_3 such that $a_1 + a_2 \equiv a_3 \pmod{q}$.

Compute $y_i = g^{a_i}$.

$$
\begin{aligned}
(v_0, v_1) &\leftarrow R^*(a_2) \\
b &\xleftarrow{\$} \{0,1\} \\
s, t &\xleftarrow{\$} \{0, ..., q-1\} \\
(\check{x}, \check{w}) &\leftarrow (g^{t'}, y_1^{t'}) \\
b' &\leftarrow R^*(\check{x}, \check{w})
\end{aligned}
$$

R^* wins if $b' = b$.

Guess

The ciphertext R^* receive contains no information about the option, so its probability of winning is equal to $1/2$.

```
module Gb' (R' : HbC) = {
  var bad : bool
  proc main() = {
    var a1, a2, a3, s, y1, ¦
    y2, y3, m0, m1, t', b, ¦
    b';
    a1        <$ dt;
    a2        <$ dt;
    a3        <- a1 + a2;
    s         <$ dt;
    y1        <- g^a1;
    y2        <- g^a2;
    y3        <- g^a3;
    (m0, m1) <@ ¦
    R'.choose(a2);
    b         <$ {0,1};
    t'        <$ dt;
    bad       <- t' = F.zero;
    b'        <@ ¦
    R'.guess(g^t', y3^t');
    return b' = b;
  }
}.

lemma Gb'_half (R' <: HbC) ¦
    &m :
  Pr[Gb'(R').main() @ &m : ¦
    res] = 1%r / 2%r.
```

Also the **Gb** game is quite similar to the one for B^*.

```
module PRFAdvR (R' : HbC) ¦
    = {
  var bad : bool
  proc main() = {
    var a1, a2, a3, s, y1, ¦
    y2, y3, m0, m1, r, t', ¦
    b, b';
    a1        <$ dt;
    a2        <$ dt;
    a3        <- a1 + a2;
    s         <$ dt;
    y1        <- g^a1;
    y2        <- g^a2;
    y3        <- g^a3;
    (m0, m1) <@ ¦
    R'.choose(a2);
    b         <$ {0,1};
    r         <@ PRFr.f(s, ¦
    b?m1:m0);
    t'        <$ dt;
    bad       <- s = F.zero;
    b'        <@ ¦
    R'.guess(g^t', y3^t'*r);
    return b' = b;
  }
}.
```

```
module PRFAdvI (R' : HbC) ¦
    = {
  var bad : bool
  proc main() = {
    var a1, a2, a3, s, y1, ¦
    y2, y3, m0, m1, r, t', ¦
    b, b';
    a1        <$ dt;
    a2        <$ dt;
    a3        <- a1 + a2;
    s         <$ dt;
    y1        <- g^a1;
    y2        <- g^a2;
    y3        <- g^a3;
    (m0, m1) <@ ¦
    R'.choose(a2);
    b         <$ {0,1};
    r         <@ ¦
    PRFi.f(b?m1:m0);
    t'        <$ dt;
    bad       <- t' = F.zero;
    b'        <@ ¦
    R'.guess(g^t', y3^t'*r);
    return b' = b;
  }
}.
```

The left game `PRFAdvR` is used to simulate the input to R, and it uses `PRFr.f` to get a function from the family. The right game `PRFAdvI` is used to simulate an input that does not contain any non-trivial information about the ballots, and it uses `PRFi` to get a random function.

The games may now be used to express the needed intermediate lemmas.

Family function
If $j = 1$ and the given function ρ is from the PRF family, this will simulate the return generator input perfectly.

```
lemma R_PRFr (R' <: HbC {INDCPA', PRFAdvR}) &m :
  `| Pr[INDCPA'(R').main() @ &m : res] - ¦
    Pr[PRFAdvR(R').main() @ &m : res] | <=
  Pr[PRFAdvR(R').main() @ &m : PRFAdvR.bad].

lemma Bad1 (R' <: HbC {PRFAdvR}) &m :
  Pr[PRFAdvR(R').main() @ &m : PRFAdvR.bad] <= 1%r / q%r.
```

Random function
If $j = N$ and the given function ρ is random, the input will contain no information about the ballots.

```
lemma Gb'_PRFi (R' <: HbC {Gb', PRFAdvI}) &m :
  `| Pr[Gb'(R').main() @ &m : res] - Pr[PRFAdvI(R').main() @ ¦
    &m : res] | <=
  Pr[PRFAdvI(R').main() @ &m : PRFAdvI.bad].

lemma Bad2 (R' <: HbC {PRFAdvI}) &m :
  Pr[PRFAdvI(R').main() @ &m : PRFAdvI.bad] <= 1%r / q%r.
```

Both these lemma's probability bounds have similar reasoning to in the intermediate lemmas in §4.2.3.

Hybrid argument
A standard hybrid argument as defined in §2.6 tells us that it is equivalent (assuming large N) to consider one and N cases. Let X_0 be the case $j = 1$ and ρ from the family, and X_1 the case that $j = N$ and ρ random. Then $\mathrm{Adv}^{\mathrm{dist}}_{X_0,X_1}(\mathcal{A}) \leq N\mathrm{Adv}^{\mathrm{dist}}_{\rho_l,\rho_{l+1}}(\mathcal{A})$ for some $k,\ 1 \leq l < N$.

```
local lemma Hybrid:
  forall &m,
    Pr[Ln(PRFb, HybGame(A)).main() @ &m : (res /\ HybOrcl.l ¦
    <= N) /\ Count.c <= 1 ] -
    Pr[Rn(PRFb, HybGame(A)).main() @ &m : (res /\ HybOrcl.l ¦
    <= N) /\ Count.c <= 1 ] =
    1%r / N%r *
     (Pr[Ln(PRFb, A).main() @ &m : (res /\ Count.c <= N) ] -
      Pr[Rn(PRFb, A).main() @ &m : (res /\ Count.c <= N) ]).
```

Using these lemmas, we can conclude with the following:

Conclusion
If R^* can extract some non-trivial information about the ballots, we have a distinguisher for the function family.

```
lemma Conclusion' (R' <: HbC {INDCPA', Gb', PRFAdvR, ¦
    PRFAdvI}) &m :
  `| Pr[INDCPA'(R').main() @ &m : res] - 1%r/2%r | <=
  `| Pr[PRFAdvR(R').main() @ &m : res] -
     Pr[PRFAdvI(R').main() @ &m : res] | + 2%r / q%r.
```

The advantage of R^* in extracting some non-trivial information about the ballots is bonded above by the advantage of a PRF-distinguisher.

The decryption service The decryption service sees the ballots in random order, and hence can extract no information about which ballot belongs to which voter.

4.3 Sketch of full protocol

The full verison of the 2010-variant of the protocol in [ste10] is sketched below.

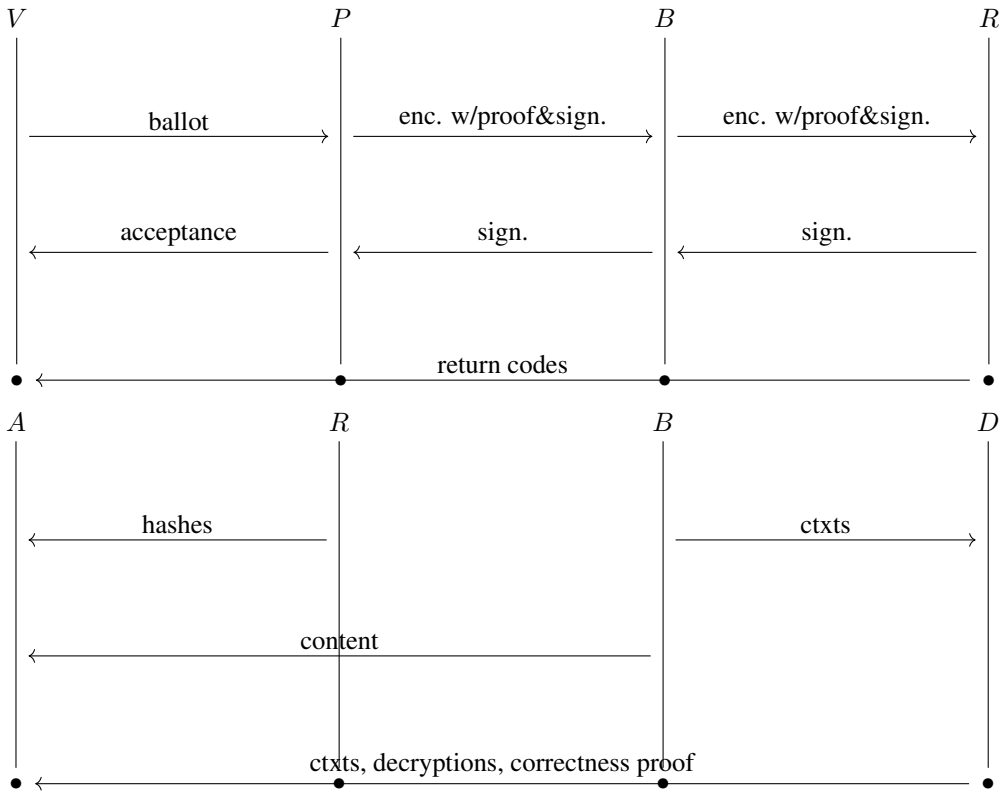


Figure 4.3: Figure 3 in 2010.

In Figure 4.3 the submission of a ballot and receiving return codes (above) and counting is illustrated.

In the ballot submission, the computer P will encrypt the voter's ballot and sign the ciphertext on the voter's behalf. It will send the ciphertext and the signature along with a proof that it knows the content of the ciphertext it encrypted to the ballot box B. After receiving this, B will compute return code ciphertexts and send them to the return code generator R along with a proof of correct computations and a signature. Then R verifiers the voter V's signature and every proof, generates the return codes and sends them directly to the voter. In addition, R signs a hash of the ballot wish it sends to B. The computer P will accept a correct signature, and inform V of acceptance.

In the counting, the decryption service D decrypts the ciphertexts it receives from B. It then shuffles the decryptions and sends them along with a proof that it is shuffles of the

ciphertexts and the ciphertexts to the auditor A. The auditor receives the entire content of B and a list of hashes of encrypted ballots from R. The auditor now verifies all the content it got.

Chapter 5

The Cryptosystem

The cryptosystem can be considered isolated from the protocol. The security properties from the cryptosystem can then be used to reason about the security of the protocol.

In the 2013 version of the Norwegian Internet Voting Protocol [ste13] a technical obstruction occurred. In 2015 a new instantiation was made in [sL15], and a Schnorr Proof of Knowledge was replaced with another Equality of Discrete Logarithms proof. The new instantiation satisfies the requirements for functionality and security, but in the security proof, the technical obstruction is avoided. The cryptosystem in this new instantiation uses the same encryption and transformation methods, and is based on the same group structure. When this new cryptosystem is used, the protocol and analysis in [ste13] still applies. The below incorporates the changes from the new instantiation.

The focus of this chapter will be on the computer P and the ballot box B.

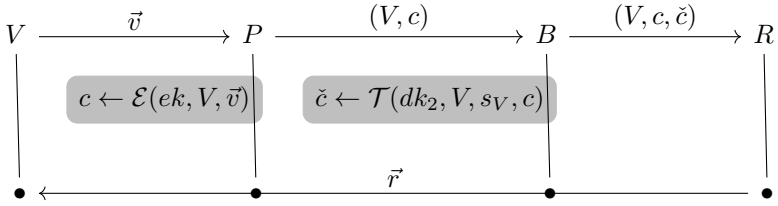


Figure 5.1: The computer P uses the encryption algorithm $\mathcal{E}$ and the ballot box B uses the transformation algorithm $\mathbb{T}$.

5.1 Definition and instantiation

Preliminaries Let I be a set of identities, M a set of messages, $O \subseteq M$ a set of options, and C a set of pre-codes. We denote options by v and ballots $(v_1, ..., v_k)$ by $\vec{v}$ $(\in O^k)$.

The cryptosystem consists of six algorithms and one protocol. The six algorithms and an additional decryption algorithm $\mathcal{D}$ that is needed only to define security requirements are instansiated below.

Key generation $\mathcal{K}$

$$
\begin{aligned}
a_{i1}, ..., a_{ik} &\xleftarrow{\$} \mathbb{Z}^*,\ i = 1, 2 \\
a_{3j} &= a_{2j} + a_{1j} \bmod q,\ 1 \le j \le k \\
y_{ij} &= g^{a_{ij}},\ i = 1, 2, 3,\ 1 \le j \le k \\
\text{choose} &\ \text{ random generator } \bar{g} \in G \\
\text{outputs keys } ek &= (g, \bar{g}, \{y_{1j}\}, \{y_{2j}\}, \{y_{3j}\}),\ 1 \le j \le k \\
\text{and } dk_i &= (a_{i1}, ..., a_{ik}),\ i = 1, 2, 3
\end{aligned}
$$

A key generation algorithm $\mathcal{K}$ that outputs a public key ek, a decryption key dk_1, a transformation key dk_2, and a pre-code decryption key dk_3.

Pre-code map generation $\mathcal{S}(ek, V)$

$$
\begin{aligned}
s &\xleftarrow{\$} \{1, ..., q-1\} \\
\gamma &= g^s \\
\text{outputs } &(s, \gamma)
\end{aligned}
$$

A pre-code map generation algorithm $\mathcal{S}$ that gets input ek and an identity V, and outputs a pre-code map s and a commitment γ to that map.

Encryption $\mathcal{E}(ek, V, \vec{v})$

$$
\begin{aligned}
t &\xleftarrow{\$} \mathbb{Z}_q \\
x &= g^t,\ \bar{x} = \bar{g}^t \\
w_i &= y_{1i}^t v_i,\ 1 \le i \le k \\
\pi_{\mathcal{E}} &\leftarrow \mathcal{P}^{I}_{eqdl}(V \parallel x \parallel w_1 \parallel ... \parallel w_k, g, \bar{g}, x, \bar{x}, t) \\
\text{outputs } c &= (V, x, \bar{x}, w_i, ..., w_k, \pi_{\mathcal{E}})
\end{aligned}
$$

An encryption algorithm $\mathcal{E}$ that gets input ek, V, and outputs a message sequence $\vec{m} \in M^k$, and outputs ciphertext c.

Extraction $\mathcal{X}(c)$

Verifies $\pi_{\mathcal{E}}$.

Computes $\bar{w} = w_1 \cdot w_2 \cdots w_k$.

Outputs $(x, \bar{w})$.

A deterministic extraction algorithm that produces the naked ciphertext $(x, \bar{w})$.

Transformation $\mathcal{T}(dk_2, V, s, c)$

$$
\begin{aligned}
&\text{verify } \pi_{\mathcal{E}} \\
&\check{x} = x^s \\
&\hat{w}_i = \check{x}^{a_{2i}},\ \check{w}_i = w_i^s,\ 1 \le i \le k \\
&\pi_{\mathcal{T}I} \leftarrow \mathcal{P}^{II}_{eqdl}(c, g, s, x, w_1, ..., w_k, \check{x}, \check{w}_1, ..., \check{w}_k) \\
&\pi_{\mathcal{T}II} \leftarrow \mathcal{P}^{III}_{eqdl}(c, g, \check{x}, a_{21}, ..., a_{2k}, \hat{w}_1, ..., \hat{w}_k) \\
&\text{outputs } \check{c} = (\check{x}, \hat{w}_1, ..., \hat{w}_k, \check{w}_1, ..., \check{w}_k, \pi_{\mathcal{T}I}, \pi_{\mathcal{T}II})
\end{aligned}
$$

A transformation algorithm $\mathcal{T}$ that gets as input dk_2, V, s and c, and outputs a pre-code ciphertext $\check{c}$ or $\perp$.

Pre-code decryption $\mathcal{D}_R(dk_3, V, \gamma, c, \check{c})$

verifies $\pi_{\mathcal{E}}, \pi_{\mathcal{T}I}, \pi_{\mathcal{T}II}$

$\rho_i = \check{w}_i \hat{w}_i \check{x}^{-a_{3i}} (= v_i^s),\ 1 \leq i \leq k$

outputs $\vec{\rho} = (\rho_1, ..., \rho_k)$

A deterministic pre-code decryption algorithm $\mathcal{D}_R$ that gets as input dk_3, V, γ, c and $\check{c}$, and outputs a sequence of pre-codes $\vec{\rho} \in C^k$ or $\perp$.

Decryption $\mathcal{D}(dk_1, V, c)$

verifies $\pi_{\mathcal{E}}$

$m_i = w_i x^{-a_{1i}},\ 1 \leq i \leq k$

outputs $\vec{m} = (m_1, ..., m_k)$

A decryption algorithm $\mathcal{D}$ that gets as input dk_1, V and c, and outputs $\vec{m} \in M^k$ or $\perp$.

A decryption protocol Π_{DP} is between a prover and a verifier. They have common input of a public key ek and a sequence of naked ciphertexts $\bar{c}_1, ..., \bar{c}_k$. The prover gets as private input a decryption key dk_1. The prover and the verifier output either $\perp$ or a sequence of messages $\vec{m}_1, ..., \vec{m}_k$.

The instantiation of Π_{DK} is not given in this paper because the focus is on the ballot submission.

Completeness requirements For the cryptosystem to be useful, it must guarantee correct decryption of ciphertexts and transformed ciphertexts. The three following completeness requirements should hold.

C1. For any message and identity, encryption followed by decryption should return the original message. For any keys ek, dk_1, any message $\vec{m}$ and identiry V, $\Pr[\mathcal{D}(ek, dk_1, \mathcal{E}(ek, V, \vec{m})) = \vec{m}] = 1$.

C2. For any sequence of messages, encrypting, extracting and then running the decryption protocol, should reproduce the messages.

C3. Transformation of a ciphertext should apply the given pre-code map to the content of the ciphertext. For any $\vec{m} \in M^k, V \in I$,
$(ek, dk_1, dk_2, dk_3) \leftarrow \mathcal{K};\ (s, \gamma) \leftarrow \mathcal{S}(ek, V);$
$c \leftarrow \mathcal{E}(ek, V, \vec{m});\ \check{c} \leftarrow \mathcal{T}(dk_2, V, s, c);$
$\vec{\rho} \leftarrow \mathcal{D}_R(dk_3, V, \gamma, c, \check{c});$
should result in $\check{c} \neq \perp$ and $\vec{\rho} = s(\vec{m})$.

5.2 Security Requirements

Security notions for the cryptosystem are sketched here.

D-Privacy Naked ciphertexts should not be correlatable to identities. For any $V \in I$ and $\vec{m} \in M^k$, if keys, ciphertext and naked ciphertext is generated as follows

$$(ek, dk_1, dk_2, dk_3) \leftarrow \mathcal{K};\ c \leftarrow \mathcal{E}(ek, V, \vec{m});\ \bar{c} \leftarrow \mathcal{X}(c),$$

then the distribution of the naked ciphertext $\bar{c}$ should be independent of the identity V.

B-Privacy An adversary that knows the transformation key dk_2 should not be able to say anything about any content of any ciphertexts she sees.

Simulator $\mathcal{S}$

$$b \xleftarrow{\$} \{0,1\}$$
$$(ek, dk_1, dk_2, dk_3) \leftarrow \mathcal{K}$$

Challenge messages requested by $\mathcal{A}$ one by one, for $i = 1, 2, ...,$

$$(\vec{m}_i^{(0)}, V_i) \leftarrow \mathcal{A}(ek, dk_2)$$
$$\vec{m}_i^{(1)} \xleftarrow{\$} M^k$$
$$c_i \leftarrow \mathcal{E}(ek, V_i, \vec{m}_i^{(b)})$$
send c_i to $\mathcal{A}$.

At any time,

$$(V, c, \check{c}, s, \gamma) \leftarrow \mathcal{A}$$
verifies that s matches γ
$$\vec{m} \leftarrow \mathcal{D}(dk_1, V, c)$$
$$\vec{\rho} \leftarrow \mathcal{D}_R(dk_3, V, \gamma, c, \check{c})$$
$$\text{send} \begin{cases} \perp, \text{ a decryption failed} \\ 1, \ (V, c) = (V_i, c_i) \\ \vec{\rho}, \text{ otherwise} \end{cases} \quad \text{to } \mathcal{A}.$$

Finally,

$$b' \leftarrow \mathcal{A}$$
$\mathcal{A}$ wins if $b' = b$

The probability of $\mathcal{A}$ winning should be close to $1/2$.

R**-Privacy** An adversary that controls the pre-code decryption key and sees many transformed encryptions of valid ballots from O^k should not be able to say anything non-trivial about the content of those encryptions.

Simulator $\mathcal{S}$

$$
\begin{aligned}
b &\xleftarrow{\$} \{0,1\} \\
\pi_1 &\xleftarrow{\$} \text{permutation on } O \\
\pi_0 &= \text{identity map on } O \\
(ek, dk_1, dk_2, dk_3) &\leftarrow \mathcal{K} \\
V &\leftarrow \mathcal{A}(ek, dk_3) \\
(s, \gamma) &\leftarrow \mathcal{S}(ek, V) \\
&\text{send } \gamma \text{ to } \mathcal{A}
\end{aligned}
$$

$\mathcal{A}$ submits sequences of ballots from O^k one by one

$$
\begin{aligned}
\vec{v}_i &\leftarrow \mathcal{A} \\
c_i &\leftarrow \mathcal{E}(ek, V, \pi_b(\vec{v}_i)) \\
\check{c}_i &\leftarrow \mathcal{T}_R(dk_2, V, s, c_i) \\
&\text{send } (c_i, \check{c}_i) \text{ to } \mathcal{A}
\end{aligned}
$$

Finally,

$$
\begin{aligned}
b' &\leftarrow \mathcal{A} \\
\mathcal{A} &\text{ wins if } b' = b
\end{aligned}
$$

The probability of $\mathcal{A}$ winning should be close to $1/2$.

A-Privacy An adversary that runs the verifier part of the decryption protocol should not be able to correlate ciphertexts with decryptions.

Simulator $\mathcal{S}$

$$
\begin{aligned}
b &\xleftarrow{\$} \{0,1\} \\
(ek, dk_1, dk_2, dk_3) &\leftarrow \mathcal{K} \\
((V_1, \vec{m}_1), ..., (V_{n'}, \vec{m}_{n'}), & \\
(V'_1, \vec{m}_1^{(0)}), ..., (V'_{n''}, \vec{m}_{n''}^{(0)})) &\leftarrow \mathcal{A}(ek) \\
\pi_0 &= \text{ identity map on } \{1, 2, ..., n'\} \\
\pi_1 &\xleftarrow{\$} \text{ permutation on } \{1, 2, ..., n'\} \\
(\vec{m}_1^{(1)}, ..., \vec{m}_{n''}^{(1)}) &\xleftarrow{\$} M^k \\
c'_i &\leftarrow \mathcal{E}(ek, V'_i, \vec{m}_i^{(b)}) \\
c_i &\leftarrow \mathcal{E}(ek, V'_i, \vec{m}_{\pi_b(i)}) \\
\bar{c}_i &\leftarrow \mathcal{X}(V_i, c_i) \\
&\text{send } (c'_1, ..., c'_{n''}, c_1, ..., c_{n'}) \text{ to } \mathcal{A} \\
&\text{run the prover part of the protocol } \Pi_{\text{DP}} \\
b' &\leftarrow \mathcal{A} \\
\mathcal{A} \text{ wins } &\text{ if } b' = b
\end{aligned}
$$

The probability of $\mathcal{A}$ winning should be close to $1/2$.

B-Integrity An adversary that knows all the key material and chooses the per-voter key material, should not be able to create an identity, a ciphertext and a transformed ciphertext such that the transformed ciphertext is inconsistent with the decryption of the ciphertext.

B-Integrity

$$
\begin{aligned}
(ek, dk_1, dk_2, dk_3) &\leftarrow \mathcal{K} \\
(V, s, \gamma, c, \check{c}) &\leftarrow \mathcal{A}(ek, dk_1, dk_2, dk_3) \\
\vec{m} &\leftarrow \mathcal{D}(dk_1, V, c) \\
\vec{\rho} &\leftarrow \mathcal{D}_R(dk_3, V, \gamma, c, \check{c})
\end{aligned}
$$

$\mathcal{A}$ wins if $\vec{\rho} \neq \perp$ and either
$\vec{m} = \perp,$ or $s(\vec{m}) \neq \vec{\rho}$.

The probability of the adversary winning should be close to 0.

D-Integrity An adversary that runs the prover's part of the protocol Π_{DP} should not be able to tamper with decryptions.

Simulator $\mathcal{S}$

$$
\begin{aligned}
(ek, dk_1, dk_2, dk_3) &\leftarrow \mathcal{K} \\
((V_1, \vec{m}_1), ..., (V_n, \vec{m}_n)) &\leftarrow \mathcal{A}(ek, dk_1) \\
c_i &\leftarrow \mathcal{E}(ek, V_i', \vec{m}_i) \\
\bar{c}_i &\leftarrow \mathcal{X}(V_i, c_i) \\
&\text{send } (\bar{c}_1, ..., \bar{c}_n) \text{ to } \mathcal{A} \\
&\text{runs the prover part of the protocol } \Pi_{\text{DP}} \\
\mathcal{A} \text{ wins } &\text{ if the verifier run outputs a sequence of messages} \\
&\quad \text{that is not a permutation of } \omega(\vec{m}_1), ..., \omega(\vec{m}_1)
\end{aligned}
$$

The probability of $\mathcal{A}$ winning should be close to 0.

5.3 Security

For the completeness requirements, we see that C1 clearly holds. C2 will not be described here since we have not described the decryption protocol.

C3. Completeness The zero knowledge proofs are complete, and ElGamal is homomorphic. The following equation will hold:

$$\rho_i = \check{w}_i \hat{w}_i \check{x}^{-a_{3i}} = w_i^s \check{x}^{a_{2i}} x^{-sa_{3i}} = (wx^{a_{2i}-a_{3i}})^s = (w_i x^{-a_{1i}})^s = v_i^s.$$

Proofs of D-privacy, B-privacy, R-privacy, A-privacy and D-integrity can be found in [ste13] and will not be described here.

The proof of B-integrity will be given in this chapter.

5.4 Equality of Discrete Logarithms

For the Equality of Discrete Logarithms (EQDL) problem in §2.10.2, we can model the protocols in EASYCRYPT. We will show the modelling of the proof in §2.10.2. The two others from §2.10.2 and §2.10.2 goes similarly.

The prover $\mathcal{P}^I_{eqdl}$ and the verifier $\mathcal{V}^I_{eqdl}$ will be modelled as follows.

```
module Prover = {
  proc i(s : aux * group * ¦
    group * group * group, ¦
    t : t) = {
    var u, a1, a2;
    u <$ dt;
    a1 = s.`2^u;
    a2 = s.`3^u;
    return (u, a1, a2);
  }
  proc iii(t u e : t) = {
    var nu : t;
    nu = u - e * t;
    return nu;
  }
}.
```

```
module Verifier = {
  proc ii(a1 a2 : group) = {
    var e;
    e <$ dt;
    return e;
  }
  proc ver(s : aux * group ¦
    * group * group * ¦
    group, a1 a2 : group, ¦
    e nu : t) = {
    return (a1 = s.`2^nu * ¦
    s.`4^e /\ a2 = s.`3^nu ¦
    * s.`5^e);
  }
}.
```

Both $\mathcal{P}^I_{eqdl}$ and $\mathcal{V}^I_{eqdl}$ will get the public input `s` $(= (aux, g, \bar{g}, x, \bar{x}))$, only $\mathcal{P}^I_{eqdl}$ will get the private input `t`.

In `i`, $\mathcal{P}^I_{eqdl}$ samples an `u` at random from `dt` $(= \mathbb{Z}_q)$ and computes `(a1, a2)` $(= (\alpha_1, \alpha_2) = (g^u, \bar{g}^u))$.

In `ii`, $\mathcal{V}^I_{eqdl}$ will get `(a1, a2)` and output an `e` chosen at random from `dt`.

In `iii`, $\mathcal{P}^I_{eqdl}$ computes `nu` with the given `t, u, e`.

In `ver` $\mathcal{V}^I_{eqdl}$ will verify that $\alpha_1 = g^\nu x^e, \alpha_2 = \bar{g}^\nu \bar{x}^e$ for given `s, a1, a2, e, nu`.

The three-way protocol run with the prover `Prover` and verifier `Verifier` is modelled below in `EQDL`.

The relation with the statement $(aux, g, \bar{g}, x, \bar{x})$ and witness t will be $x = g^t$ and $\bar{x} = \bar{g}^t$.

Step I $\mathcal{P}^I_{eqdl}$ chooses random $u \in \mathbb{Z}_q$ and computes $\alpha_1 = g^u, \alpha_2 = \bar{g}^u$ and sends (α_1, α_2) to $\mathcal{V}^I_{eqdl}$.

Step II $\mathcal{V}^I_{eqdl}$ chooses random challenge $e \in \mathbb{Z}_q$ and sends to $\mathcal{P}^I_{eqdl}$.

Step III $\mathcal{P}^I_{eqdl}$ computes $\nu = u - et \bmod q$ and sends to $\mathcal{V}^I_{eqdl}$.

$\mathcal{V}^I_{eqdl}$ accepts iff $\alpha_1 = g^\nu x^e, \alpha_2 = \bar{g}^\nu \bar{x}^e$.

Completeness

A proof by a honest $\mathcal{P}^I_{eqdl}$ will be accepted by $\mathcal{V}^I_{eqdl}$.

```
op Relation (s : aux * ¦
    group * group * group ¦
    * group, t : t)
  = (s.`4 = s.`2^t /\ s.`5 ¦
    = s.`3^t).

module EQDL = {
  proc main(s, t) = {
    var  u, a1, a2, e, nu, ¦
    b;
     (u, a1, a2) <@ ¦
    Prover.i(s, t);
     e <@ Verifier.ii(a1, ¦
    a2);
     nu <@ Prover.iii(t, u, ¦
    e);
     b <@ Verifier.ver(s, ¦
    a1, a2, e, nu);
     return b;
  }
}.

lemma Completeness &m (s : ¦
    aux * group * group * ¦
    group * group) (t : t) :
  Pr[EQDL.main(s, t) @ &m ¦
    : Relation s t => res] ¦
    = 1%r.
```

The lemma `Completeness` states that the probability of the verifer accepting the proof must be equal to one if the relation holds.

To prove that the protocol is SHVZK, we will need a simulator which will be modelled as follows.

Given any challenge e, we can choose a random ν and compute $\alpha_1 = g^{\nu}x^{e}, \alpha_2 = \bar{g}^{\nu}\bar{x}^{e}$ with x and $\bar{x}$ as above and get α_1, α_2 with the same distribution as in the real proof. We denote this sampling by $\nu \leftarrow Sim^{I}_{eqdl}(aux, g, \bar{g}, x, \bar{x})$.

```
module SIM = {
  proc sim_eqdl(s : aux * ¦
    group * group * group ¦
    * group, e : t) = {
    var nu;
    nu <$ dt;
    return nu;
  }
  proc main(s, e) = {
    var nu, a1, a2;
    nu <@ sim_eqdl(s, e);
    a1 = s.`2^nu * s.`4^e;
    a2 = s.`3^nu * s.`5^e;
    return (a1, a2);
  }
}.
```

Using this simulator, we can play a game and prove equivalence between a real proof and a simulated proof.

Given challenge e, we have a left game that uses the prover.

We also have a right game that uses the simulator described above, with given e.

SHVZK

(α_1, α_2) from the simulator will have the same distribution as in the real proof by the prover.

```
module SHVZK_L = {
  proc main(s, t, e) = {
    var u, a1, a2, nu;
    (u, a1, a2) <@ ¦
    Prover.i(s, t);
    nu <@ Prover.iii(t, u, ¦
    e);
    return (a1, a2);
  }
}.

module SHVZK_R = {
  proc main(s, e) = {
    var a1, a2;
    (a1, a2) <@ ¦
    SIM.main(s, e);
    return (a1, a2);
  }
}.

equiv SHVZK s t e :
  SHVZK_L.main ~ ¦
    SHVZK_R.main : arg{1} ¦
    = (s, t, e) /\ arg{2} ¦
    = (s, e) ==> Relation ¦
    s t => ={res}.
```

The lemma `SHVZK` states that the left game using the prover and the right game using the simulator are equivalent. As long as the relation holds, the proofs will have the same distribution and be indistinguishable. Note that only the prover gets the witness, the simulator only gets the statement.

5.5 NIZK

5.5.1 Random Oracle

In EASYCRYPT, the random oracle is modelled as a finite mapping, and the random function samples values uniformly at random. On the same queries, the random oracle will always output the same value.

Random oracles

I

$H_2 : \{0,1\}^* \times G^6 \to \{1,...,2^\tau\}$

$e \leftarrow H_2(aux, g, \bar{g}, x, \bar{x}, \alpha_1, \alpha_2)$

II

$H_1 : \{0,1\}^* \times G^{2k+2} \to \{1,...,2^\tau\}^k$

$H_2 : \{0,1\}^* \times G^{2k+4} \to \{1,...,2^\tau\}$

$\vec{e} \leftarrow H_1(aux, g, \gamma, \vec{w}, \vec{\hat{w}})$

$e \leftarrow H_2(aux, g, \gamma, \vec{w}, \vec{\hat{w}}, \alpha_1, \alpha_2)$

III

$H_1 : \{0,1\}^* \times G^{2k+2} \to \{1,...,2^\tau\}^k$

$H_2 : \{0,1\}^* \times G^{2k+4} \to \{1,...,2^\tau\}$

$\vec{e} \leftarrow H_1(aux, g, \check{x}, \vec{y_2}, \vec{\hat{w}})$

$e \leftarrow H_2(aux, g, \check{x}, \vec{y_2}, \vec{\hat{w}}, \alpha_1, \alpha_2)$

```
module H : H = {
  var m1 : (stm, out list) ¦
    fmap
  var m2 : (inp, out) fmap
  proc init() = { m1 <- ¦
    empty; m2 <- empty; }

  proc h1(s : stm) : out ¦
    list = {
    var evec : out list;
    evec <$ dlist dout k;
    if (! s \in m1)
      m1.[s] <- evec;
    return oget (m1.[s]);
  }
  proc h2(i : inp) : out = {
    var e : out;
    e <$ dout;
    if (! i \in m2)
      m2.[i] <- e;
    return oget (m2.[i]);
  }
}.
```

This version of the random oracle is a lazy one, that is, it does not populate the key-value store before it is queried. To initialize the random oracle, the `init()`-function is used to set the key-value store to empty. Adversaries, provers and verifiers are only given access to the random function, not the initializing.

5.5.2 Proof of correct computations

I When the computer P is encrypting, we want to make sure it computes the ciphertext c correctly. This will be done by proving that two element have the same discrete logarithms relative to disitinct generators g and $\bar{g}$ of the group G and the proof is between a prover and a verifier, as in §2.10.2.

The bad event is that the adversary $\mathcal{A}$ queries the random oracle with a statement for which the relation does not hold, and it results in a valid hash value.

Lemma 2.10.1 in §2.10.2 is stated in EASYCRYPT as

```
lemma dout1E: forall (x : out), mu1 dout x = 1%r / (2^τ)%r.
```

and Lemma 2.10.3 as

```
axiom Bad1 :
  mu (dlist dout k) inVdim'k_1 <= 1%r / (2^τ)%r.
```

The probability of $\mathcal{A}$ to forge a proof $\pi_{\mathcal{E}}$ (the two other proofs $\pi_{\mathcal{T}I}$ and $\pi_{\mathcal{T}II}$ have similar lemmas) is stated as

```
lemma SoundAI (A <: I.ANY {I.Sys.SYS, I.Sys.Bound, I.Sys.H})
      &m (bad : (glob I.Sys.H) -> bool) :
  (forall m2 m1, bad (m2, m1) = E2I m2) =>
  Pr[I.SoundA(A, I.Sys.H).main() @ &m : bad (I.Sys.H.m2, ¦
    I.Sys.H.m1)] <= qH%r/ (2^τ)%r.
```

and Theorem 2.10.2 is expressed as

```
lemma Sound (A <: I.ANY {I.Sys.SYS, I.Sys.Bound, I.Sys.H}) &m :
  phoare[A(Bound(H)).forge : Bound.c = 1 ==> Bound.c <= qH] = ¦
    1%r =>
  phoare[A(Bound(H)).forge : true ==> ! Rel res.`1 witness] = ¦
    1%r =>
  hoare[A(Bound(H)).forge : true ==> (res.`2.`1 = (\o d/D) <=>
                                      (Inp' res.`1 (oget ¦
    H.m1.[res.`1]) res.`2 \in H.m2))] =>
  phoare[A(Bound(H)).forge : true ==> res.`2.`1 = (\o d/D)] <= ¦
    (qH%r/(2^τ)%r) =>
  Pr[Sound(A).main() @ &m : res] <= (2 * qH)%r / (2^τ)%r.
```

Lemma `dout1E` states that for all values sampled at random from the uniform distribution `dout` $(= \{1, ..., 2^\tau\})$, the probability of sampling a specific value is $1/2^\tau$. Lemma `Bad1` (here stated as an axiom) states that the probability of sampling a vector that lies inside the supspace V defined in §2.10.3 is at most $1/2^\tau$.

When the ballot box B is encrypting, we want to make sure that it computes the transformed ciphertext $\check{c}$ correctly. For this, we will need to prove that several group elements are raised to the same power of a certain element, and that several elements has been correctly raised to distinct powers.

II The proof is between a prover and a verifier as in §2.10.2.
The bad event for H_2 is similar to the one in I, and the proof goes the same way.

The bad event for H_1 is that $\Pi_{i=1}^k \check{w}_i^{e_i} = (\Pi_{i=1}^k w_i^{e_i})^s$.

III The proof is between a prover and a verifier as in §2.10.2.
The bad event for H_2 is similar to the one in I, and the proof goes the same way.

The bad event for H_1 is that there exists an a such that $\Pi_{i=1}^k y_{2i}^{e_i} = g^a$ and $\Pi_{i=1}^k \hat{w}_i^{e_i} = \check{x}^a$.

With the Fiat-Shamir transformation from §2.9, the proofs can be made non-interactive. Both the prover and the verifier have access to a random oracle `H` $(= \mathcal{H})$ with functions `h1` $(= H_1)$ and `h2` $(= H_2)$, as in §5.5.1.

The prover $\mathcal{P}_{eqdl}^I$'s and the verifier $\mathcal{P}_{eqdl}^I$'s algorithms for the non-interactive proofs are modelled in EASYCRYPT as follows.

Prover $\mathcal{P}^I_{eqdl}(aux, g, \bar{g}, x, \bar{x}, t)$

$u \xleftarrow{\$} \mathbb{Z}_q$
$\alpha_1 \leftarrow g^u$
$\alpha_2 \leftarrow \bar{g}^u$
$e \leftarrow H_2(aux, g, \bar{g}, x, \bar{x}, \alpha_1, \alpha_2)$
$\nu \leftarrow u - et \bmod q$
return $\pi_{\mathcal{E}} = (\nu, e)$

```
module (P : Prover) (H : ¦
    AH) = {
  proc prove(s : stm, w : ¦
    wit) = {
    var u, evec, e, nu;
    u <$ dt;
    evec <@ H.h1(s);
    e <@ H.h2(Inp s evec u);
    nu <- u - ((\*) (\q e) ¦
    evec w);
    return (e, nu);
  }
}.
```

Verifier $\mathcal{V}^I_{eqdl}(aux, g, \bar{g}, x, \bar{x}, \pi_{\mathcal{E}})$

$(\nu, e) \leftarrow \pi_{\mathcal{E}}$
$\alpha_1 \leftarrow g^{\nu} x^e$
$\alpha_2 \leftarrow \bar{g}^{\nu} \bar{x}^e$
$e' \leftarrow H_2(aux, g, \bar{g}, x, \bar{x}, \alpha_1, \alpha_2)$
return $e \stackrel{?}{=} e'$

```
module (V : Verifier) (H : ¦
    AH) ={
  proc verify(s : stm, p : ¦
    prf) = {
    var evec, e, nu, e';
    (e, nu) <- p;
    evec    <@ H.h1(s);
    e'      <@ H.h2(Inp' s ¦
    evec p);
    return (e = e');
  }
}.
```

A proof for which the relation holds, must be accepted by the verifier.

Completeness

The random function is initialized.

It is checked if the relation holds. If it does not, output $\perp$.

If it holds, generate a proof from an honest prover.

Verify the proof with an honest verifier.

Output the answer of the verifier.

A proof for a statement for which the relation holds, generated by an honest prover, must be accepted by an honest verifier with probability one.

```
module Completeness(R : ¦
    Relation, P : Prover, ¦
    V : Verifier, H : H) = {
  proc main(s : stm, w : ¦
    wit) : bool = {
    var p, b;
    H.init();
    SYS.rel <@ R.main(s, w);
    b <- false;
    if (SYS.rel) {
      p <@ P(H).prove(s, w);
      b <@ V(H).verify(s, ¦
    p);
    }
    return b;
  }
}.
lemma Complete (s' : stm, ¦
    w' : wit) &m :
  Pr[Completeness(R, P, V, ¦
    H).main(s', w') @ &m : ¦
    SYS.rel => res] = 1%r.
```

The simulator used to simulate the real protocol is as following.

For **I** , $stm = (aux, g, \bar{g}, x, \bar{x})$.

For **II** , $stm = (aux, g, \gamma, \vec{w}, \vec{\tilde{w}})$.

For **III** , $stm = (aux, g, \tilde{x}, \vec{y_2}, \vec{\tilde{w}})$.

$Sim_{eqdl}(stm, \vec{e}, e)$

$$\nu \xleftarrow{\$} \{1, ..., 2^{\tau}\}$$

Output ν.

Simulator

$$e \xleftarrow{\$} \{1, ..., 2^{\tau}\}$$
$$\vec{e} \leftarrow H_1(stm)$$
$$\nu \leftarrow Sim_{eqdl}(stm, \vec{e}, e)$$

Reprogram H_2.

The proof is (e, ν).

```
module (S : Simulator, H) ¦
    = {

module Sim = {
  proc main(s : stm, evec ¦
    : out list, e : out) : ¦
    t = {
    var nu : t;
    nu <$ dt;
    return nu;
  }
}
  var inp : inp

  proc init = H.init

  proc h1 = H.h1

  proc h2(i : inp) : out = {
    var e : out;
    e <$ dout;
    return e;
  }

  proc prove(s : stm) : ¦
    prf = {
    var evec, e, nu;
    e    <@ h2(Inp s ¦
    witness<:out list> ¦
    witness<:t>);
    evec <@ h1(s);
    nu <@ Sim.main(s, ¦
    evec, e);
    inp <- Inp' s evec (e, ¦
    nu);
    (* Reprogram h2(inp) = ¦
    e : *)
    H.m2.[inp] <- e;
    return (e, nu);
  }
}.
```

The simulator does not have access to the random oracle H_2 for querying, but will reprogram it to output the randomly sampled value at the given query.

We will have an abstract zero knowledge adversary with adversary to the hash functions H_1 and H_2.

The adversary $\mathcal{A}^{H_1,H_2}$ has access to query H_1 and H_2. It outputs a statement and a witness. For a given proof for the statement, it outputs a bit.

An axiom stating that when the abstract relation holds, the input to the hash oracle can be calculated in the prover and the verifier to give the same input. This abstraction will later be instantiated for the three proofs of correct computation, **I** , **II** and **III** .

```
module type AdvZK (H : AH) ¦
    = {
  proc a1()                ¦
    : stm * wit {H.h1 H.h2}
  proc a2(p' : prf option) ¦
    : bool       {H.h1 H.h2}
}.

axiom RelInp s w evec u (p ¦
    : out * t) :
  Rel s w => Inp s evec u ¦
    = Inp' s evec (p.`1, ¦
    p.`2).
```

As defined in the theory, a left and a right ZK game will be used to bound the advantage of the adversary in distinguish between the simulator and the real proof. The two games are as follows.

ZKL

Initialize logs and random oracles.

$$(s, w) \leftarrow \mathcal{A}(Log(H))$$
$$\pi \leftarrow \mathcal{P}(H)(s, w)$$
$$b \leftarrow \mathcal{A}(Log(H))(\pi)$$

ZKR

Initialize logs and random oracles.

$$(s, w) \leftarrow \mathcal{A}(Log(H))$$
$$\pi \leftarrow \mathcal{S}(s)$$
$$b \leftarrow \mathcal{A}(Log(H))(\pi)$$

```
module ZK_L(R: Relation, P ¦
    : Prover, A : AdvZK, H ¦
    : H) = {
  proc main(): bool = {
    var p;
    Log(H).init();
    (SYS.s, SYS.w) <@ ¦
    A(Log(H)).a1();
    SYS.rel <@ R.main ¦
    (SYS.s, SYS.w);
      p      <@ ¦
    P(H).prove(SYS.s, ¦
    SYS.w);
      SYS.p <- Some p;
    SYS.b <@ ¦
    A(Log(H)).a2(SYS.p);
    return SYS.b;
  }
}.

module ZK_R(R: Relation, ¦
    S:Simulator, A:AdvZK, ¦
    H : H) = {
  proc main(): bool = {
    var p;
    Log(S).init();
    (SYS.s, SYS.w) <@ ¦
    A(Log(H)).a1();
    SYS.rel <@ R.main ¦
    (SYS.s, SYS.w);
      p      <@ ¦
    S.prove(SYS.s);
      SYS.p <- Some p;
    SYS.b <@ ¦
    A(Log(H)).a2(SYS.p);
    return SYS.b;
  }
}.
```

In both **ZKL** and **ZKR,** first the random oracles and the logs are initialized, then the adversary is given access to query it, and eventually outputs a statement s and a witness w. In **ZKL,** the game proceeds with the real prover. The prover has access to the random oracle, and generates a proof π which is given to the adversary. The adversary will now try to guess if she has received a real proof or a simulated one. In **ZKR,** the adversary is given a simulated proof π and makes a similar try on distinguishing. Note that the simulator never receives the witness w the adversary outputs.

A bound for the adversary being able to distinguish between the two is given below.

Non-interactive Zero Knowledge
Given an adversary $\mathcal{A}$ that outputs a statement and a witness such that the relation holds, the advantage of $\mathcal{A}$ in distinguishing between the simulated proof and the real one, is bounded by the probability of $\mathcal{A}$ guessing the query such that it is in the set of logged queries.

```
lemma ZK (A <: AdvZK {H, S, Log, SYS}) &m :
  hoare[A(Log(H)).a1 : true ==> Rel res.`1 res.`2] =>
  Pr[ZK_L(R, P, A, H).main() @ &m : res] <=
  Pr[ZK_R(R, S, A, H).main() @ &m : res] + Pr[ZK_R(R, S, A, ¦
    H).main() @ &m : S.inp \in Log.qs2].
```

5.5.3 Soundness

It must be hard to generate valid proofs when the discrete logarithms are not equal, that is, the relation does not hold.

An algorithm that has access to a random oracle will output a statement for which the relation does not hold and a proof as follows.

ANY

I

$$(aux, g, \bar{g}, x, \bar{x}, \pi_{\mathcal{E}}) \leftarrow \mathcal{A}(\mathcal{H}^{\eta})$$

$\mathcal{A}$ wins if $\log_g x \neq \log_{\bar{g}} \bar{x}$ and $1 \leftarrow \mathcal{V}^{I}_{eqdl}(aux, g, \bar{g}, x, \bar{x}, \pi_{\mathcal{E}})$.

II

$$(aux, g, \gamma, \vec{w}, \vec{\check{w}}, \pi_{\mathcal{T}I}, s) \leftarrow \mathcal{A}(\mathcal{H}^{\eta})$$

$\mathcal{A}$ wins if $w_i^s \neq \check{w}_i$ for some i, and $1 \leftarrow \mathcal{V}^{II}_{eqdl}(aux, g, \gamma, \vec{w}, \vec{\check{w}}, \pi_{\mathcal{T}I})$.

III

$$(aux, g, \check{x}, \vec{y_2}, \vec{\hat{w}}, \pi_{\mathcal{T}II}, \vec{a_2}) \leftarrow \mathcal{A}(\mathcal{H}^{\eta})$$

$\mathcal{A}$ wins if $y_{2i} = g^{a_{2i}} \forall i$, but $\check{x}^{a_{2i}} \neq \hat{w}_i$ for some i and $1 \leftarrow \mathcal{V}^{III}_{eqdl}(aux, g, \check{x}, \vec{y_2}, \vec{\hat{w}}, \pi_{\mathcal{T}II})$.

```
module type ANY (H : AH) = {
  proc forge() : stm * prf ¦
    {H.h1 H.h2}
}.

module Sound (A : ANY) = {
  proc main() = {
    var s, p, evec;
    Bound(H).init();
    (s, p) <@ ¦
    A(Bound(H)).forge();
    (SYS.s, SYS.p) <- (s, ¦
    Some p);
    SYS.b <- false;
    if (!(SYS.s \in H.m1)) {
      evec <@ H.h1(SYS.s);
      if (!(Inp' SYS.s ¦
    evec (oget SYS.p)) \in ¦
    H.m2) {
        SYS.b <@ ¦
    V(H).verify(SYS.s, ¦
    oget SYS.p);
      }
    }
  }
}.
```

We have an adversary algorithm `ANY` that is used in a soundness-game `Sound` together with a verifier `V`.

Looking at all the tree proofs of correct computations from §2.10.2, we have an adversary that outputs relevant statements for which the relation does not hold. The adversary wins if the corresponding verifier verifies the forged proof.

The point the adversary outputs may either not be in the domain of the random oracle (i.e. the adversary has not queried it at the relevant point) and will be verified by the verifier by a small probability (the resulting randomly sampled value equals the one outputted by she adversary), or it may be a forged proof which will be verified by the verifier with probability equal to one.

Theorem 3.1
I: Take any algorithm that outputs a proof $\pi_{\mathcal{E}} = (\nu, e)$, a bit string aux, and group elements $g.\bar{g}, x, \bar{x}$ such that $\log_g x \neq \log_{\bar{g}} \bar{x}$. If the algorithm uses at most η queries to the random oracle H_1, then the probability that

$$\mathcal{V}^{I}_{eqdl}(aux; g, \bar{g}; x, \bar{x}; \pi_{\mathcal{E}})$$

is at most $2\eta/2^{\tau}$.
The theorems for **II** and **III** are similar.

```
lemma Sound (A <: ANY {SYS, Bound, H}) &m (bad : (glob H) -> ¦
    bool) m1A m2A :
  hoare[H.init : true ==> ! bad (glob H)] =>
  hoare[H.h1 : bad (glob H) ==> bad (glob H)] =>
  hoare[H.h2 : bad (glob H) ==> bad (glob H)] =>
  phoare[H.h1 : ! bad (glob H) ==> bad (glob H)] <= (1%r / ¦
    (2^τ)%r) =>
  phoare[H.h2 : ! bad (glob H) ==> bad (glob H)] <= (1%r / ¦
    (2^τ)%r) =>
  phoare[A(Bound(H)).forge : Bound.c = 0 ==> Bound.c <= qH] ¦
    = 1%r =>

  hoare[A(Bound(H)).forge : true ==> H.m1 = m1A /\ H.m2 = ¦
    m2A] =>

  Pr[Sound(A).main() @ &m : bad (m2A, m1A) \/ SYS.b] <= 2%r ¦
    * qH%r / (2^τ)%r.
```

The soundness game in EASYCRYPT uses the adversary algorithm and the verifier algorithm. First, the adversary has η tries with a probability $1/2^{\tau}$ to forge a proof, and then it may output a forged proof or a proof for which it has not queried the hash oracle. This proof is given to the verifier, which will accept with probability $1/2^{\tau}$ if it is an unqueried point, or with probability 1 if the proof is forged.

5.5.4 B-Integrity

One of the security notions for the cryptosystem is *B-Integrity*. The computer P and the ballot box B are corrupt and cooperate. The computer uses the encryption algorithm $\mathcal{E}$ to compute the ciphertext c and generate a proof of correct computation $\pi_{\mathcal{E}}$. The decryption algorithm $\mathcal{D}$ verifies this proof when it is to decrypt c. The ballot box uses the transformation algorithm $\mathcal{T}$ to produce a ciphertext $\check{c}$ and proof of correct computations $\pi_{\mathcal{T}I}$ and $\pi_{\mathcal{T}II}$ which the pre-code decryption algorithm $\mathcal{D}_R$ verifies when it decrypts $\check{c}$.

For the integrity property in the ballot box B, a game is played between an adversary $\mathcal{A}$ and a simulator $\mathcal{S}$.

B-Integrity

$(ek, dk_1, dk_2, dk_3) \leftarrow \mathcal{K}$

$(V, s, \gamma, c, \check{c}) \leftarrow \mathcal{A}^{\mathcal{H}^\eta}(ek, dk_1, dk_2, dk_3)$

$\vec{m} \leftarrow \mathcal{D}(dk_1, V, c)$

$\vec{\rho} \leftarrow \mathcal{D}_R(dk_3, V, \gamma, c, \check{c})$

$\mathcal{A}$ wins if $\vec{\rho} \neq \perp$ and either $\vec{m} = \perp$, or $s(\vec{m}) \neq \vec{\rho}$.

```
module type ADV (HI : ¦
    I.Sys.H, HII : ¦
    II.Sys.H, HIII : ¦
    III.Sys.H) = {
  proc a(ek : group list * ¦
    group list * group list,
         dk1 dk2 dk3 : t ¦
    list) : I * t * group ¦
    * c * cc (* V, s, ¦
    gamma, c, cc *) {HI.h1 ¦
    HI.h2 HII.h1 HII.h2 ¦
    HIII.h1 HIII.h2}
}.
module Sim_lite (A : ADV) ¦
    = {
  proc main() = {
  var ek, dk1, dk2, dk3;
  var v, s, gamma, c, cc;
  var m, rho;
  I.Sys.Bound(I.Sys.H) ¦
    .init();
  II.Sys.Bound(II.Sys.H) ¦
    .init();
  III.Sys.Bound(III.Sys.H) ¦
    .init();
  (ek, dk1, dk2, dk3) <@ ¦
    KG.kg();
  (v, s, gamma, c, cc) <@ ¦
    A(I.Sys.Bound(I.Sys.H), ¦
    II.Sys.Bound(II.Sys.H), ¦
    III.Sys.Bound(III.Sys.H)) ¦
    .a(ek, dk1, dk2, dk3);
  m <@ Dec_lite.dec(dk1, ¦
    v, c);
  rho <@ ¦
    DecR_lite.dec(dk3, v, ¦
    gamma, c, cc);
  return (rho, m, s);
  }
}.
```

Decryption

$\mathcal{D}(dk_1, V, c)$

Verifies $\pi_{\mathcal{E}}$

Computes $m_i = w_i x^{-a_{1i}},\ 1 \leq i \leq k$

Outputs $\vec{m} = (m_1, ..., m_k)$

if the proof verifies, or

$\perp$ otherwise

```
module Dec_lite = {
  proc dec(dk1 : t list, v ¦
    : I, c : c) = {
  var x, x', w, p, m;
  (v, x, x', w, p) <- c;
  m <- Some (zW (fun w (a ¦
    : t), w * xˆ(-a)) w ¦
    dk1);
  return m;
  }
}.
```

Pre-Code Decryption

$\mathcal{D}_R(dk_3, V, \gamma, c, \check{c})$

Verifies $\pi_{\mathcal{E}}, \pi_{\mathcal{T}I}, \pi_{\mathcal{T}II}$

Computes $\rho_i = \check{w}_i \hat{w}_i \check{x}^{-a_{3i}}$
$(= m_i^s),\ 1 \leq i \leq k$

Outputs $\vec{\rho} = (\rho_1, ..., \rho_k)$
if the proofs verifies, or
$\perp$ otherwise

```
module DecR_lite = {
  var inpII : II.Sys.inp
  var inpIII : III.Sys.inp
  var bII : bool
  var bIII : bool
  proc dec(dk3 : t list, v ¦
    : I, gamma : group, c ¦
    : c, cc : cc) = {
  var auxI, auxII, auxIII ¦
    : bits;
  var g, x, x', w, pI, xc, ¦
    wh, wc, pII, pIII, rho;
  var evecII, evecIII;
  (v, x, x', w, pI) <- c;
  (xc, wh, wc, pII, pIII) ¦
    <- cc;
  auxI <- encodeI (v, x, ¦
    x', w);
  auxII <- encodeII c;
  auxIII <- encodeIII c;
  g <- Top.g;
  bII <- false;
  bIII <- false;
  if ((auxII, g, gamma, w, ¦
    wc) \notin ¦
    II.Sys.H.m1) {
     evecII <@ ¦
    II.Sys.H.h1((auxII, g, ¦
    gamma, w, wc));
      inpII <- II.Sys.Inp' ¦
    (auxII, g, gamma, w, ¦
    wc) evecII pII;
    if (inpII \notin ¦
    II.Sys.H.m2) {
      bII <@ ¦
    II.Sys.V(II.Sys.H) ¦
    .verify((auxII, g, ¦
    gamma, w, wc), pII);
    }
  }
```

Continued below.

```
  if ((auxIII, g, xc, ¦
    KG.y2, wh) \notin ¦
    III.Sys.H.m1) {
    evecIII <@ ¦
    III.Sys.H.h1((auxIII, ¦
    g, xc, KG.y2, wh));
    inpIII <- III.Sys.Inp' ¦
    (auxIII, g, xc, KG.y2, ¦
    wh) evecIII pIII;
    if (inpIII \notin ¦
    III.Sys.H.m2) {
      bIII <@ ¦
    III.Sys.V(III.Sys.H) ¦
    .verify((auxIII, g, ¦
    xc, KG.y2, wh), pIII);
    }
  }
  rho <- None;
  if (bII /\ bIII) {
    rho <- Some (zW2 G.( * ¦
    ) (fun w (a : t), w * ¦
    xc^(-a)) wc wh dk3);
  }
  II.Sys.SYS.b <- bII;
  III.Sys.SYS.b <- bIII;
  return rho;
  }
}.
```

B-Integrity
An adversary that knows all the key material and chooses the per-voter key material, should not be able to create an identity, a ciphertext and a transformed ciphertext such that the transformed ciphertext is inconsistent with the decryption of the ciphertext.
Adversary wins if $\vec{\rho} \neq \perp$ and either $\vec{m} = \perp$, or $s(\vec{m}) \neq \vec{\rho}$.
The probability of the adversary winning should be close to 0.

```
lemma B_Int (A <: ADV {SYS, KG, HSYS})
    (AII <: II.ANY {II.Sys.SYS, II.Sys.Bound, II.Sys.H})
    (AIII <: III.ANY {III.Sys.SYS, III.Sys.Bound, ¦
    III.Sys.H}) &m gHII gHIII
    (badII : (glob II.Sys.H) -> bool) (badIII : (glob ¦
    III.Sys.H) -> bool)
    (ek' : group list * group list * group list) (dk1' dk2' ¦
    dk3' : t list) :
  (forall m2 m1, badII (m2, m1) = (E2II m2 \/ E1II m1)) =>
  (forall m2 m1, badIII (m2, m1) = (E2III m2 \/ E1III m1)) =>
  hoare[KG.kg : true ==> res = (ek', dk1', dk2', dk3')] =>
  hoare[A(I.Sys.Bound(I.Sys.H), II.Sys.Bound(II.Sys.H), ¦
    III.Sys.Bound(III.Sys.H)).a : true ==> gHII = ¦
    (II.Sys.H.m2, II.Sys.H.m1) /\ gHIII = (III.Sys.H.m2, ¦
    III.Sys.H.m1)] =>
  hoare[B_IntA(A).main : true ==> gHII = (II.Sys.H.m2, ¦
    II.Sys.H.m1) /\ gHIII = (III.Sys.H.m2, III.Sys.H.m1)] =>
  Pr[Sim_lite(A).main() @ &m :
     let (rho, m, s) = (res.`1, res.`2, res.`3) in
     rho <> None] <= (4 * qH)%r / (2^τ)%r.
```

Since both $\mathcal{D}$ and $\mathcal{D}_R$ checks the $\pi_{\mathcal{E}}$-proof, it cannot be the case that $\vec{\rho} \neq \perp$ and $\vec{m} = \perp$. The case then reduces to the probability of both $\pi_{\mathcal{T}I}$ and $\pi_{\mathcal{T}II}$ being verified. The probability of the adversary succeeding in forging at least one of the proofs is bounded by $4\eta/2^{\tau}$.

Chapter 6

The Voting Protocol

The full protocol from 2013 is described in [ste13], and is an improvement of the simplified one in [ste10] described in Chapter 4. This version uses multi-ElGamal and more efficient NIZK proofs for a performance improvement.

The security of the protocol follows from the security of the cryptosystem.

In this chapter an overview of the full protocol from 2013 will be given. The voting protocol will be sketched, including the environment assumptions and the phases, before an overview of the security analysis is given.

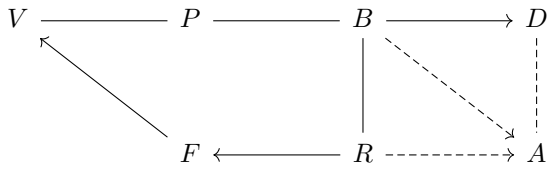


Figure 6.1: Full protocol. Figure 1 in 2013.

Figure 6.1 is a sketch of the players involved in the protocol, and their communication. The four players to the right, B, R, D and A, are the infrastructure players. The voter V submits his ballot to a computer P, which encrypts the ballot and submits it to the ballot box B. The ballot box and a return code generator R computes return codes for the ballot and sends them to the voter's cell phone F. When the ballot box closes, the submitted encrypted votes are decrypted by a decryptor D while an auditor A supervises.

In the protocol, there is a *setup phase,* a *ballot submission phase,* and a *counting phase.* The players in the protocol are divided into three categories: the voters with their computers and phones, the infrastructure players, and the electoral board members. It is assumed to be secure, authenticated channels between the infrastructure players, the voters and computers, the voters and their phones, and a one-way channel from the return code generator to the

voter's phones. The return code generator will have a signing key, and the ballot box and the computers will have the corresponding verification keys. All other keys are assumed generated by a trusted dealer.

The distribution of the key material to the different players are as follows

V The set $\{(m, D_V(s_V(m))) \mid m \in O\ \{1_O\}\}$.

P Public key ek.

B Transformation key dk_2 and $\{(V, s_V)\}$.

R Pre-code decryption key dk_2 and $\{(V, \gamma_V, D_V)\}$.

D Decryption key dk_1.

A Public key ek.

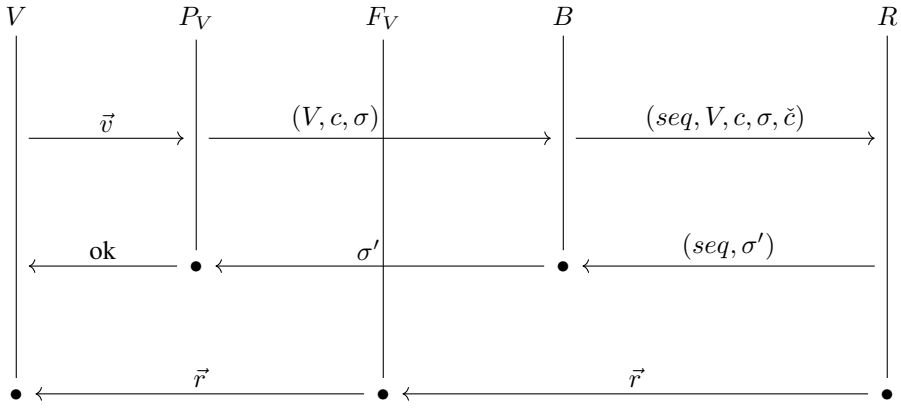


Figure 6.2: Full protocol. Figure 10 in 2013.

During the ballot submission the messages in Figure 6.2 are sent. The voter is V, her phone is F_V and the computer she uses to vote is P_V.

Setup phase The key generation may happen a long time before the election, so it is not important that the generation is quick. During this setup phase, only the electoral board- and infrastructure-players are active.

Ballot submission phase During the ballot submission, the voter's with their computers and phones, along with two of the infrastructure players (the ballot box and the return code generator), are active. First, the voter gives her ballot to a computer. The computer then

encrypts the ballot, and submits it, along with a signature on the voter's behalf, to the ballot box. Then the ballot box transforms the ciphertext and send everything to the return code generator. The return code generator creates the return codes which it sends to the voter's phone. In addition, it signs a hash of the ballot and sends the signature to the ballot box. The ballot box stores the encrypted ballot and sends the return code generator's signature to the voter's computer. Then the computer verifies the signature and tells the voter if the ballot was accepted or not. The voter receives the return codes from her phone, and the voter accepts if the computer accepted the ballot and the return codes are correct.

The ballot submission ends when the ballot box is told to close.

Counting phase During the counting, only infrastructure players are active. The ballot box is told to close, and after this it waits until ongoing ballot submissions are done, but refuses to accept any new submissions. When the ballot box closes, it informs the return code generator that it has closed, and sends every recorded ballot to the auditor. Then it extracts naked ciphertext from the valid ballots, and sends them to the decryptor.

The return code generator send all its records to the auditor, and the decryptor informs the auditor which naked ciphertexts it received, before running the decryption protocol. While running the decryption protocol, the auditor works as a verifier. It also verifies that the ballot box and the return code generator agree on which ballots were submitted, and that the ballot box and the decryptor agrees on the valid naked ciphertexts that should be counted. Lastly, the auditor runs the decryption protocol with the decryption as a prover.

6.1 Security analysis

Ideal functionality is a protocol where there is a trusted party that communicates over secure channels with the protocol participants and can compute the desired protocol output, or if it is a protocol indistinguishable from an ideal functionality. This can not be realized under adaptive corruption for public key encryption. Since the cryptosystem is essentially public key encryption, corruption is restricted to static.

Static corruption is when the set of corrupted players is chosen before the election, and is fixed.

In the setup phase, the electoral board and the infrastructure players generate keys. If an adversary blocks key generation, functionality never enters the ballot submission.

In the ballot submission phase, the adversary may interfere with ballot submissions to a certain degree, or attempt forgery of the ballots. The amount of information that leaks directly to the adversary, is denoted by a $leak$-function defined as

$$leak(\Lambda, V, P, \vec{m}) = \begin{cases} \vec{m}, \ P \text{ corrupt} \\ \Lambda(\vec{m}), \ R \text{ corrupt} \\ \perp, \ \text{otherwise,} \end{cases}$$

where Λ is a permutation of the option set. That means a corrupt computer will learn the voter's chosen options, and a corrupt return code generator will learn a permutation of it.

In the counting phase, an adversary may interfere to a certain degree.

The protocol is only guaranteed security if no more than one infrastructure player is corrupt. Here we have four categories of who is statically corrupt:

- The ballot box, a subset of the voters and a subset of the computers
- The return code generator
- The decryptor
- The auditor

The security of the internet voting protocol follows from the security of the cryptosystem, that is, the security of the encryption scheme and properties of certain infrastructure.

Chapter 7

Concluding remarks

The goal of this paper has been to use a code-based sequence of games approach to reason about security components of the cryptosystem underlying the Norwegian Internet Voting Protocol.

Further work includes more thorough analysis of the constructions, in addition to formalizing and verifying other components of the security proof, including privacy and integrity notions.

Bibliography

[cat17] catalindragan. Easycrypt code for privacy of labelled-minivoting. *https://github.com/catalindragan/minivoting-privacy/tree/master/proof*, 2017.

[sL15] Kristian Gjøsteen and Anders Smedstuen Lund. The norwegian internet voting protocol: A new instantiation. *Cryptology ePrint Archive, Report 2015/503*, 2015.

[ste10] Kristian Gjøsteen. Analysis of an internet voting protocol. *Cryptology ePrint Archive, Report 2010/380*, 2010.

[ste13] Kristian Gjøsteen. The norwegian internet voting protocol. *Cryptology ePrint Archive, Report 2013/473*, 2013.

[War17] Cortier Dragan Dupressoir Schmidt Strub Warinschi. Machine-checked proofs of privacy for electronic voting protocols. *https://www.ieee-security.org/TC/SP2017/papers/401.pdf*, 2017.