Formally Verified Verifiable Electronic Voting Scheme

Mukesh Tiwari

A thesis submitted for the degree of
Doctor of Philosophy at
The Australian National University

February 2021

Except where otherwise indicated, this thesis is my own original work.

Mukesh Tiwari
23 February 2021

To my grandparents who — despite being poor and uneducated —
understood the value of education.

Acknowledgments

Going through this PhD has truly been a remarkable experience. It has taught me many valuable lessons, but the most important one is to not get intimidated by a hard problem. During this journey, I was fortunate to have guidance and support from many people.

First and foremost, this thesis could not have been possible without the support of my supervisor, Dirk Pattinson. I really admire his abilities and intuition to make sure that I stay clear from many dead ends. Moreover, I thank him for patiently answering my stupid questions happily and guiding me towards the answer by asking the right questions. My only wish to be a researcher like him and incorporate more of his qualities, but I am less optimistic about my chance.

I also want to thank my dog, Turbo, whom I can talk for hours, and he never judged me. He would patiently listen to my proofs and ideas about electronic voting with occasionally barking at me if he was bored of listening. His listening helped me in developing my ideas more better. Turbo truly made my PhD a breeze, and I never felt any pressure of PhD when I was with him. Thank you Turbo, and you are the best dog.

I want to thank Optus, Australian Mobile Service Provider, for unlimited calling hours scheme to India. Because of this scheme, I was able to talk to my family, specially my mom, everyday for hours. She is from the generation who has seen the mobile and internet revolution in their late age and had hard time in coping up with technological advancement.

Some of the great friends who made this journey possible are:

- Caitlin D'Abrera: She is one the smartest person I know of. She has a super power to break down any problem at finer details to have a better understanding. Moreover, her curiosity to understand the *git* tool forced me to learn it properly, so that I can answer her all the queries. Always be curious Caitlin.
- Milad Ketabi Ghale Ali: Milad was a senior PhD student at the logic

group when I started my PhD, but we have not had any interaction until I was in my 8th month of PhD. But once we come to know each other, it mostly turned out to be discussing Immanuel Kant, Ludwig Wittgenstein, David Hume, and Rumi. One of his favourite thing during any discussion was to not let anyone speak, hide the argument or not mention it all, and later bring it to end the discussion. In the beginning, it was frustrating, but later we (me and Caitlin) learnt to enjoyed it because it was always a fruitful discussion and chance to know some good philosophical theories.

- Ali Cheraghian: One of the most funniest person who can lighten up any heavy atmosphere with his stupid jokes. One of the most sought up skills of Ali is to make everyone excited about disclosing something that we did not know, but he would withhold it at the very last moment driving everyone crazy.

- Jim De Groot: The funny Dutch who is really very good at making jokes in all the situations. I also admire to be a Category theorist like him, but I do not see it happening in near future. He is too much into exploring nature and because of him, I got a chance to explore the nature around Canberra.

- Ian Shillito: The philosopher who did not mind my philosophy jokes. Sometimes, I had very profound discussion about the philosophy itself with him.

- Saeed Alhamlan: Saeed was a business student when I started my PhD. I came to know him through Milad. I travelled to Sydney with Saeed in my new year Holidays, and it was a memorable trip. He is great at making travelling plans and a excellent travelling companion.

Many thanks to my co-supervisors Rajeev Goré and Michael Norrish. Their valuable feedback during my PhD monitoring helped me a lot in shaping my presentation skills. Also, I attended programming language reading group every Wednesday with Michael, and as usual, he was always flawless in explaining ideas.

I want to thank Thomas Haines, my co-author, for teaching me all the bits pieces of cryptography, specifically zero-knowledge-proofs. I really enjoyed working with him and hope to produce many more papers in the upcoming future.

One of the best experience of this PhD was travelling to Princeton to participate in the DeepSpec summer and attending the lectures of Coq stalwarts like Andrew Appel, Benjamin Pierce, and Adam Chlipala. I have learned plenty of tricks about the Coq by watching their lectures on the YouTube. Now that I understand dependent types enough, but it was not always the case. In order to dissect the dependent types, I was watching Benjamin Pierce's Coq lecture on the YouTube. During his lecture he said "proof objects are just data structures", and it was precisely my aha moment in the path of demystifying the dependent types. Also, it was fun to meet many denizens of #coq internet relay chat (irc) denizens in person at the DeepSpec summer school, who patiently answered my numerous questions about dependent types. Unfortunately, there were some sad moments as well. I could not attend Marktoberdorf Summer School because it was next to impossible for me to get the visa of Germany, but I hope things would be more easier in the upcoming future.

I want to thank the Australian National University for providing me resources and scholarship to attend conferences and explore the academic world. In addition, many thanks to Inger Mewburn and her team at ANU for organizing the thesis boot camp which was very helpful in understanding the thesis writing. If, during this thesis, I am not able to convey something in a meaningful way, then the fault is mine because I could not incorporate their teachings properly during the thesis boot camp.

No thesis can be complete in Australia without mentioning the Australian beers and wines. Australia has some of the best beers and wines, and they were very helpful at many occasions during this PhD journey.

Finally, one of the biggest joy of this PhD was to meet my girlfriend, Mina, who came as a visitor to our logic group. Being by your side for the last one and half years has been the best thing that ever happened to me, and I could not imagine a life without you. I am also grateful to all the efforts you made to bear my tendencies to constantly talk about set-theory, formal-verification, and every thing except romantic talks. Thank you for being my side all the time, Mina.

Notations

- AMD - Advance Micro Devices
- CoC - Calculus of Construction
- CIC - Calculus of Inductive Construction
- PTS - Pure Type System
- ZKP - Zero-Knowledge Proof
- $\mathbb{N}$ - Set of Natural Numbers
- $\mathbb{Z}$ - Set of Integers
- IACR - International Association for Cryptologic Research
- AEC - Australian Electoral Commission
- DES - Data Encryption Standard
- EVM - Electronic Voting Machine
- GCC - GNU Compiler Collection
- FMUL - Floating Point Multiplication
- RTL - Register Transfer Language
- HOL - Higher Order Logic
- LEDA - Library for Efficient Data Types and Algorithms
- LCF - Logic for Computable Functions
- PVS - Prototype Verification System
- ACL2 - A Computational Logic for Applicative Common Lisp
- NSA - National Security Agency
- MITM - Man-in-the-Middle

- SSL - Secure Socket Layer
- DL - Discrete Logarithm
- LFP - Least Fix Point
- GFP - Greatest Fix Point
- RAM - Random Access Memory

Abstract

Since the introduction of secret ballots in Victoria, Australia in 1855, paper (ballots) are widely used around the world to record the preferences of eligible voters. Paper ballots provide three important ingredients: correctness, privacy, and verifiability. However, the paper ballot election brings various other challenges, e.g. it is slow for large democracies like India, and error prone for complex voting method like single transferable vote, and poses operational challenges for large countries like Australia. In order to solve these problems and various others, many countries are adopting electronic voting. However, electronic voting has a whole new set of problems. In most cases, the software programs used to conduct the election have numerous problems, including, but no limited to, counting bugs, ballot identification, etc. Moreover, these software programs are treated as commercial in confidence and are not allowed to be inspected by members of the public. As a consequence, the result produced by these software programs can not be substantiated.

In this thesis, we address the three main concerns posed by electronic voting, i.e. correctness, privacy, and verifiability. We address the correctness concern by using theorem prover to implement the vote counting algorithm, privacy concern by using cryptography, and verifiability concern by generating a independently checkable scrutiny sheet (certificate). Our work has been carried out in the Coq theorem prover.

Contents

List of Figures

Chapter 1

Introduction

The best weapon of a dictatorship is secrecy, but the best weapon of a democracy should be the weapon of openness.

Niels Bohr

1.1 Problem Statement

A democracy can be best described as a system where all eligible voters have equal rights to express their opinion(s) on different matters. One of the most important example of expressing opinion is by holding election to elect the leader of country. During the election, all eligible voters express their opinion on a paper, known as ballot, in a manner, depending on the voting method, which reflects their true intention. For example, if the voting method is *ranked voting (preferential voting)*, the voters rank the candidates according to their preference, and if the method is *first past the post*, each voter selects one candidate by marking against the candidate name on the ballot. Later, once the ballot cast finishes, a candidate is elected as a winner from the participating candidates by combing the choices of all the voters. The paper ballot method works great, except it is very time consuming, expensive, error prone, and not very inclusive for disabled voters such as the visually impaired. In order to solve the various problems posed by paper ballot, many countries are adopting electronic voting as an alternative. Electronic voting is getting popular in many countries, and the reason for its popularity is cost-effective, faster result, high voter turn out, and accessible for disabled voters. Undeniably, electronic voting has helped, for example, Australia to ease the logistic challenges of elections because of its massive land size and sparse population and save millions of dollars. In addition, it has helped India, the second most populous country with 900 million eligible voter, to declare 2019 election with 67 percent voter turn out (roughly 600 million) in 2 days, and Estonia, a labour shortage country, has saved thousands of man hours, 11,000 working days, by using electronic voting [Est].

Despite all these benefits, electronic voting is an arduous effort because a minuscule possibility of going anything wrong in software or hardware could lead to an undesirable situation [Lewis et al.], [Halderman and Teague, 2015], [Aranha et al., 2019], [Feldman et al., 2007]. The nature of (electronic) data and the ease of its manipulability/misinterpretation causes electronic voting many problems, which are not present in paper ballot elections, that makes it perfectly susceptible to delivering wrong and unverifiable results [Wolchok et al., 2010]. For instance, if a software program used in electronic voting for reading the ballots has byte order bug, or even if it depends on some other software which has byte order bug (the data is supposed to read from left to right, but software is reading right to left), then the interpretation of a ballot would be completely different from what the voter had in mind. More often than not, these software programs are configured incorrectly [Kohno et al., 2004] and run at the top of (untrusted) operating systems and hardware. Usually, operating systems have millions of lines of code (for example, Linux has 15 millions lines) which exposes a large attack surface and could be exploited, possibly by the current government or foreign countries, for illegal gain. The worst, these software and hardware are commercial in confidence and treated as a black-box, and, most often, their source code or design is not open for public scrutiny [Australian Electoral Commission, 2013]. In addition, these software programs take a list of ballots and produce the result without producing any evidence about the correctness of result. As a consequence, from casting the ballot electronically to declaring winner based on the cast (electronic) ballot, the whole process lacks basic assumptions of democracy such as transparency, genuineness, and verifiability.

In order to make the electronic voting process genuine and trustworthy, the electronic voting research community has recognised some must-have properties of electronic voting protocol [Küsters et al., 2011], [Benaloh and Tuinstra, 1994], [Delaune et al., 2010a], [Bernhard et al., 2017]:

- Correctness: The produced results are correct and convincing to all leaving no ground for suspicion.

- Coercion-resistance: A voter can not cooperate with a coercer to prove anything about her choices.

- Eligibility: Only eligible voters can cast a ballot.

- Privacy: All the votes must be secret, and a voter should not be able to convince anyone the value of her vote.

- End-to-end Verifiability: Any independent third party should be able to verify the final outcome of election based on cast ballots. It can be further divided into three sub-categories:

  - Cast-as-intended: Every voter can verify that their ballot was cast as intended.
  - Collected-as-cast: Every voter can verify that their ballot was collected as cast.
  - Tallied-as-cast: Everyone can verify the final result on the basis of the collected ballots.

In this thesis, we focus on privacy, correctness, coercion-resistance, and tallied-as-cast, the third part of end-to-end verifiability, property of an election. Furthermore, we assume that the first two properties of end-to-end verifiability, cast-as-intended and collected-as-cast, hold for an election. Cast-as-intended is a verification method that is used to audit the front end voting software, also known as voting client software, to make sure that it is not modifying the options of voters. In a nutshell, the cast-as-intended is assurance to a voter that front-end software is transparent and her vote is recorded according to her intent. Cast-as-intended is an active area of research in its own right [Galindo et al., 2015], [Marky et al., 2018], [Cortier et al., 2019]; however, it is not the focus of this thesis. Similarly, collect-as-cast is a notion related to the voters to make sure that the ballots appearing on the bulletin board are indeed the ballots that cast during the election. A consequence of collected-as-cast notion is that any attempt to change or delete the ballots from the bulletin board would be detected. This notion is indeed a crucial one and works as a bridge between the cast-as-intended and tallied-as-cast notions. However, it is related to voters' behaviour; hence, the reason for assumption. Moreover, assuming these two notions, cast-as-intended and collected-as-cast, helped us in isolating the irrelevant details and paved a way to focus more on the complex problem of counting, i.e. tallied-as-cast.

1.2 Research Motivation and Contribution

Given the potential advantages of electronic voting, we need to address the correctness, privacy, and verifiability concerns for its widespread adoption. This thesis sets out to address these concerns of electronic voting. The questions we asked ourselves were:

1. Can we implement a vote counting protocol with a "guarantee" (maximum possible assurance that we can get about software programs with respect to some specification) that the resulting implementation is correct and practical enough to count millions of ballots in a real-life election (Correctness)?

2. Can we produce the result by counting encrypted ballots without revealing its content, and at the same time, assuring everyone that the result produced is only based on "valid" ballots, and "invalid" ones have been discarded (Privacy and Coercion-resistance)?

3. Can we decouple the verifiability from the implementation details of a vote counting software program, i.e. generating enough evidence so that any independent auditor can ascertain the outcome of an election without trusting the implementation of the vote counting software program used to conduct the election (Verifiability)?

In order to answer these questions, at first we need two things: (i) a voting protocol and (ii) a tool to implement the voting protocol and prove the correctness properties of the implementation. Our choice of voting protocol is the Schulze method [Schulze, 2011] and the tool is Coq [Bertot et al., 2004] theorem prover for implementing and proving the correctness of the Schulze method. Even though the Schulze method is not used in any democratic election to public office, it is one of the most popular method to elect candidates for various organisations, e.g. Debian, GnuPG, KDE, etc., over the Internet and political groups, e.g. pirate party of Australia, Belgium, Brazil, Germany, etc[1]. One of the major reason for its popularity is that it has many desirable properties. While no preferential voting scheme can guarantee all desirable properties that one would like to impose due to Arrow's theorem [Arrow, 1950a], the Schulze method offers a good compromise, with a number of important properties already established in Schulze's original paper. Amongst the various properties, the Schulze method satisfies the *resolvability criterion*,

[1] https://en.wikipedia.org/wiki/Schulze_method#Users

i.e. it elects a single winner under the assumption that number of voters are much larger than number of candidates (and in case of a tie, when there is more than one winner, a random vote can be selected to declare the winner. However, our formalisation has not taken the randomness into account, so it can produce more than one winner).

Coq is a theorem prover (proof assistant) based on the *Calculus of Inductive Construction (CIC)* [Coquand and Huet, 1988] [Coquand and Paulin, 1988]. The calculus of inductive construction is a highly expressive formal system (type system) which allows "proof" terms and "computation" terms to live in the same universe (level). Moreover, during the proof development, it provides step by step feedback to the user and the possibility to automate proofs by writing custom tactics using the Ltac [Delahaye, 2000]. In addition, Coq proofs can be extracted into the Haskell, OCaml, and Scheme.

Now that we have the voting protocol (Schulze method) and the tool (Coq theorem prover), we demonstrate that it is possible to achieve correctness, privacy, coercion-resistance, and (tallied-as-cast) verifiability in electronic voting. We achieve the following:

- *Correctness* by formally specifying the Schulze method and prove its correctness properties inside the Coq theorem prover. Coq has a well-developed extraction facility that we use to extract proofs into OCaml programs, and using these extracted OCaml programs, we have counted the ballots from an election to produce the result.
- *Privacy and Coercion-resistance* by encryption. We use homomorphic encryption to compute the final tally without decrypting any individual ballot. The encryption hides the preference of voters, facilitating ballot privacy and preventing any possible coercion.
- *Verifiability* by tabulating the relevant data of an election (which we call the scrutiny-sheet/certificate). Achieving verifiability in a plaintext ballot counting is fairly straightforward. To achieve verifiability in encrypted ballot counting, we augment the scrutiny sheet with zero-knowledge proofs for each claim we make during the counting, which can later be checked by any auditor.

In addition to demonstrating correctness, privacy, and verifiability, we have also developed a formally verified certificate checker. Moreover, we have shown that our implementation adheres to the various properties established by Schulze in his original paper.

Formally Verified Checker: Third party independent election audit based on scrutiny sheet data is a crucial step towards establishing the trust in the system. However, auditing the scrutiny sheet of an election involving encrypted ballots is not straightforward in comparison to an election with plaintext ballots. In general, auditing the scrutiny sheet of an election involving plaintext ballots simply requires the knowledge of basic arithmetic, e.g. addition, subtraction and multiplication, and virtually anyone can audit the election based on the data produced in the scrutiny sheet by using a calculator or by writing a simple program in her preferred language. However, an encrypted ballot election scrutiny sheet involves various cryptographic concepts (homomorphic encryption, zero-knowledge proof, commitment scheme, etc.) which are accessible to very few voters, mainly cryptographers, so auditing it requires a deep understanding of cryptographic principals. To ease this situation, we have developed a formally verified certificate checker as a proof of concept for automating the audit of an election, conducted on encrypted ballots. Having said that, our certificate generated by encrypted ballots is very complex, and formalizing all the cryptographic primitives involved would be fairly time consuming, so we have developed a proof of concept formally verified certificate checker for the International Association of Cryptologic Research (IACR) 2018 election scrutiny sheet (the IACR scrutiny sheet is relatively simple compared to our certificate).

Properties of Schulze Method: We have proved two properties, Condercet winner and Reversal symmetry amongst many, of the Schulze method inside the Coq theorem prover (ongoing work). These properties could be seen as an ultimate stress testing for an implementation, and we have shown that our implementation of the Schulze method follows two important properties, i.e. Condercet winner and Reversal symmetry. Ideally, we would like to prove that our implementation follows all the properties established in the Schulze's paper [Schulze, 2011].

1.3 Cryptographic Blackbox

Since the beginning of this project, our primary goal was to achieve privacy (using encryption) and verifiability (using zero-knowledge proof) in electronic voting using cryptographic primitives (but not the verification of primitives itself). To achieve this goal, we have taken the axiomatic approach and assumed the existence of cryptographic primitives inside Coq. Moreover, we assume axioms about their correctness behaviour, e.g. decryption is left inverse of encryption. These primitives, in general, provide functionality of generating a random permutation, encrypting a plaintext data, decrypting a ciphertext data, producing commitment of a value, constructing a zero-knowledge proof, and verifying a zero-knowledge proof. Later, in extracted OCaml code from Coq code, these functions are instantiated with Unicrypt [Locher and Haenni, 2014] functions[2].

1.4 Publication

The chapters, or some part of it, of this thesis are based on the following papers:

1. Pattinson D., Tiwari M. (2017) Schulze Voting as Evidence Carrying Computation. In: Ayala-Rincón M., Muñoz C. (eds) Interactive Theorem Proving. ITP 2017. Lecture Notes in Computer Science, vol 10499. Springer, Cham. https://doi.org/10.1007/978-3-319-66107-0_26

2. Bennett Moses L., Goré R., Levy R., Pattinson D., Tiwari M. (2017) No More Excuses: Automated Synthesis of Practical and Verifiable Vote-Counting Programs for Complex Voting Schemes. In: Krimmer R., Volkamer M., Braun Binder N., Kersting N., Pereira O., Schürmann C. (eds) Electronic Voting. E-Vote-ID 2017. Lecture Notes in Computer Science, vol 10615. Springer, Cham. https://doi.org/10.1007/978-3-319-68687-5_5

3. Ghale M.K., Goré R., Pattinson D., Tiwari M. (2018) Modular Formalisation and Verification of STV Algorithms. In: Krimmer R. et al. (eds) Electronic Voting. E-Vote-ID 2018. Lecture Notes in Computer Science, vol 11143. Springer, Cham. https://doi.org/10.1007/978-3-030-00419-4_4

4. Haines T., Pattinson D., Tiwari M. (2020) Verifiable Homomorphic Tallying for the Schulze Vote Counting Scheme. In: Chakraborty S., Navas J. (eds) Verified Software. Theories, Tools, and Experiments. VSTTE 2019. Lecture Notes in Computer Science, vol 12031. Springer, Cham. https://doi.org/10.1007/978-3-030-41600-3_4

[2]Formalising the whole cryptographic stack used in our project would be very time consuming (probably a PhD itself), but it would be worth trying. Although, we have formalised the (ElGamal) encryption, and decryption inside Coq, but we still are very far from achieving the goal of fully verified cryptographic stack. We leave the formalisation of cryptographic primitives for future work (work in progress).

5. Thomas Haines, Rajeev Goré, and Mukesh Tiwari. 2019. Verified Verifiers for Verifying Elections. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19). Association for Computing Machinery, New York, NY, USA, 685–702. DOI:https://doi.org/10.1145/3319535.3354247

Part of chapter 2 is based on *No More Excuses: Automated Synthesis of Practical and Verifiable Vote-Counting Programs for Complex Voting Schemes*, chapter 4 is based on *Schulze Voting as Evidence Carrying Computation*, chapter 5 is based on *Verifiable Homomorphic Tallying for the Schulze Vote Counting Scheme*, and part of chapter 6 is based on *Verified Verifiers for Verifying Elections*.

1.5 Related Work

There is extensive work that addresses the different issues related of electronic voting protocols in a symbolic model (pi-calculus [Milner, 1999] [Abadi and Fournet, 2001], a formal language to describe and analyse the process), but there are very few, to the best of my knowledge, that have used theorem provers to implement the voting protocol (counting algorithm) and verify its correctness properties. Pi-calculus has been used by [Kremer and Ryan, 2005] and [Delaune et al., 2010b] to model and analyse the various properties, such as fairness, eligibility, vote-privacy, receipt-freeness and coercion-resistant, of the protocol FOO developed by [Fujioka et al., 1993]. A general technique to model the remote electronic protocol and automatically verify its security properties using pi-calculus has been put forward by [Backes et al., 2008]. Moreover, pi-calculus is used by [Cortier and Smyth, 2011] to analyse the ballot secrecy of [Helios, 2016]. Similarly, [Cortier and Wiedling, 2012] have used pi-calculus to ascertain properties of the Norwegian electronic voting protocol. Receipt-freeness and vote-privacy of the Selene voting protocol [Ryan et al., 2016] have been proved by [Bruni et al., 2017] using Tamarin [Meier et al., 2013]. Most of these works differ from ours in the sense that their primary focus is verification of security protocols in Dolev-Yao model [Dolev and Yao, 1983], whereas our work is more focused on verified implementation and the verifiability aspect of vote counting.

The closest to our work are [Cochran and Kiniry, 2010] [DeYoung and Schürmann, 2012], [Pattinson and Schürmann, 2015], [Verity et al.], [Verity and Pattinson, 2017], and [Ghale et al., 2017]. Business Object Notation (BON) and Java Modelling Language (JML) have been used by [Cochran and Kiniry, 2010] to formally specify the Java implementation of Irish Proportional Representation by Single Transferable Vote (PR-STV) method. They relied on Extended Static Checking to validate the correctness of their implementation. Upon further investigation [Cochran and Kiniry, 2013], they improved it by writing formal specification of candidate, ballot, and ballot box datatypes using the Alloy model checker [Jackson, 2002]. However, they themselves pointed out that:

> Note that this automated consistency checking is not the same as providing a full interactive proof of a soundness theorem in a higher-order logical framework. Such formalisation is an interesting and useful exercise, but we did not do it for this case study. Instead, checking the dozens of theorem stipulated in law text is more akin to the kind of validation that we are advocating in this work. It gives us high confidence, but not a proof, that the mechanical formalization is sound and complete.

Linear logic [Girard, 1987] has been used by [DeYoung and Schürmann, 2012] to model the different entities in electronic voting as a resource. The use of linear logic makes it very natural to capture the different entities in electronic voting, depending on their usage, by means of modality, e.g. a voter can cast only one vote, but she might need to show her photo ID multiple times at the counting booth. Mathematical proof theory has been used by [Pattinson and Schürmann, 2015] to treat the vote counting as a mathematical proof, and in the same vein, [Ghale et al., 2017] have formalised single transferable vote in Coq and extracted Haskell code from the formalisation. The extracted Haskell code produces the result and a certificate for a given set of input ballots. This certificate can be used by any third party to verify or audit the outcome of the election result. In further research, [Ghale et al., 2018] developed a formally verified certificate checker using the theorem prover HOL4 [Slind and Norrish, 2008]. Moreover, they connected the HOL4 proofs to the formally verified compiler CakeML [Kumar et al., 2014] to get an executable which was correct with respect to the formal specification of the protocol down to machine level. However, none of these works consider privacy and coercion resistance as a key issue in electronic voting, and their method simply works for plaintext ballots which are susceptible to "Italian" attack [Otten, 2003] [Benaloh et al., 2009]. In a nutshell the "Italian" attack can be described as follows:

> a full disclosure of ballots in preferential voting system carries the potential danger of ballot identification of a particular voter if the

number of candidates participating in election is large. Suppose that 40 candidates are participating in an election, then there are *40! (815915283247897734345611269596115894272000000000)* complete preference options and many more incomplete preference options (if it is allowed) for a voter to fill her ballot. Since the number of options is very large, if a candidate and a voter want to collude, then the candidate would ask the voter to mark her first and every other candidate in a certain order (an unique permutation). Later, once the ballots are published on the bulletin board, then the unique permutation can be used by the candidate to identify the vote of each voter.

1.6 Outline of the Chapters

- Chapter 2 provides an overview of electronic voting around the world, problems in general, and rationale for formal verification of election voting software.

- Chapter 3 provides the overview of concept of Coq theorem prover and cryptographic primitives.

- Chapter 4 describes the Schulze method, its formal specification, proof of correctness, experimental results, and scrutiny sheet.

- Chapter 5 describes verifiable homomorphic tally for the Schulze method, its realisation in the Coq theorem prover, experimental results, instructions to audit the scrutiny sheet.

- Chapter 6 focuses on the notion of software independence, and sketches details for the formalisation of cryptographic concepts involved in the certificate generated by encrypted ballots.

- Chapter 7 puts forward the idea of machine checked properties of electronic voting schemes and describes a couple of the properties, Condercet winner and reversal symmetry, of the Schulze method.

- Chapter 8 concludes the thesis, and some possible direction of future work.

1.7 Trivia

Before 1856, Victoria and NSW held their elections to elect its democratic representative in pubs where it was legal for candidates to offer beer to voters to influence their decision!

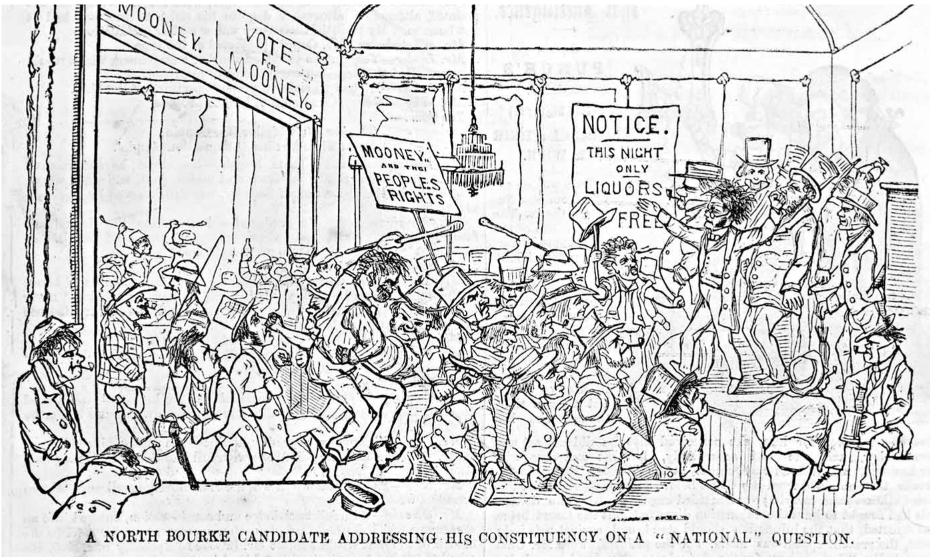

Figure 1.1: Election held in 1855 in Victoria, Australia was conducted in pub!

Chapter 2

Background

People shouldn't be afraid of their government. Governments should be afraid of their people.

Alan Moore, V for Vendetta

Counting ballots by hand is a tedious, error prone, slow, and costly process. For example, the Senate election conducted in Western Australia in September 2013 was declared void by the high court because of the loss of 1370 votes. It was re-conducted in April 2014 at the cost of 20 Million AUD with additional delay in results [Aus]. Before introduction to electronic voting machines in India, it used to take months to declare the result. As a consequence, many countries are now adopting electronic voting to alleviate the problems introduced by hand counting. The world can be divided into five broad categories according to the usage of electronic voting [Evo] (Figure 2.1): i) No electronic voting (Grey Area), ii) Discussion and/or voting technology pilots (Yellow Area), iii) Discussion and concrete plans for Internet voting (Orange Area), iv) Ballot scanners, Electronic Voting Machines, and Internet Voting (Green and Dark Green), v) Withdrawn voting technology because of public concern (Red Area).

Chapter Outline: In section 2.1, we discuss the major concern in electronic voting, bugs in the software/hardware, by high lighting the state of electronic voting in Australia, Germany, India, and Netherlands. In the following section 2.2, we give two anecdotes that how formal verification helped in achieving the correctness and eliminating bug in CompCert [Leroy, 2006], a formally verified C compiler, and Athelon, a microprocessor designed by Advanced Micro Devices (AMD), with the emphasis that we should formally

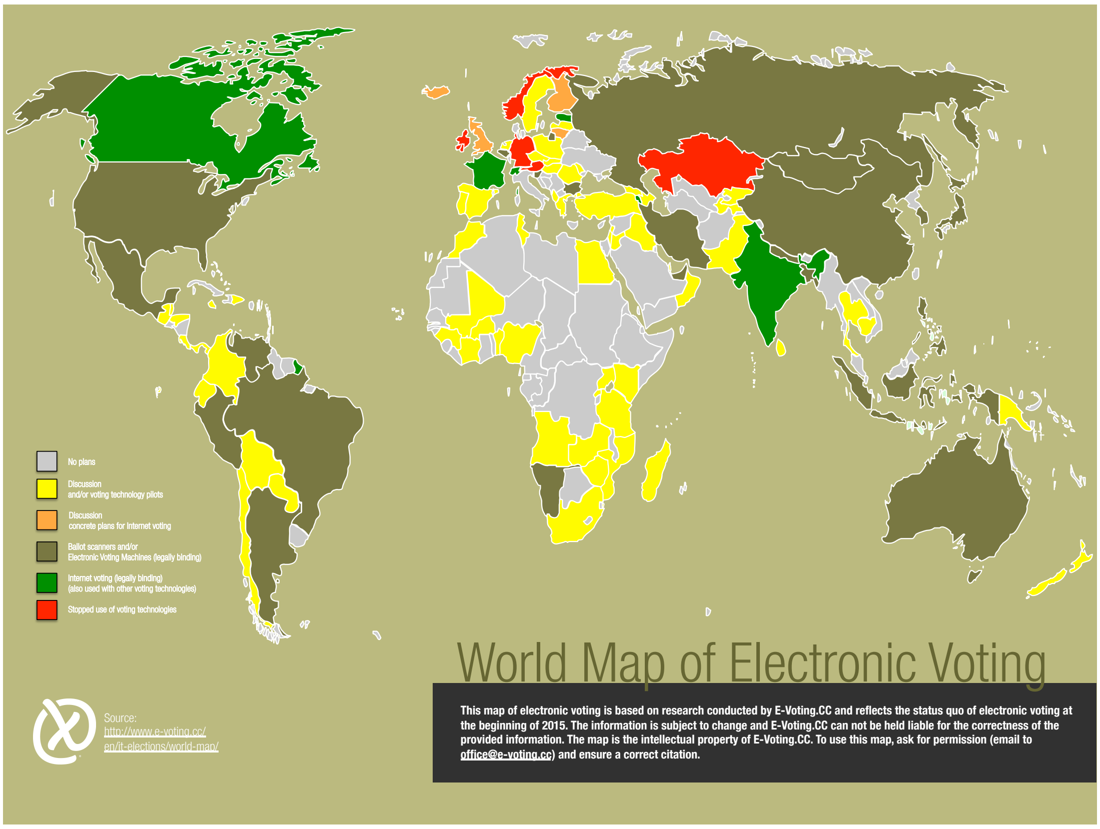

Figure 2.1: World map of Electronic Voting (source: https://www.e-voting.cc/en/it-elections/world-map/)

verify the software programs used in electronic voting to alleviate the concerns of software/hardware bug. Section 2.3 emphasizes that formal verification alone is not enough to establish the trust and puts forward the concept of scrutiny sheet (2.3.1), which can be used independently to attest the result of an election, to achieve verifiability. Finally, we summarize in the section 2.5 by emphasizing that formal verification and verifiability, both are needed to ensure the trust in electronic voting.

2.1 Electronic Voting

Electronic voting is projected as a step towards the future with many benefits, such as increased voter turnout, faster result, accessible to everyone including challenged voters, and reduced carbon footprint (for each national election, India saves about 10,000 tonnes of the ballot paper by using electronic voting machines). There is no doubt that electronic voting has many advantages over paper ballots, but it is certainly not flawless. Electronic voting makes the process faster, but it has its own layer of added complexities which creates trust issues amongst voters. For this reason, some countries who were the early adopters were also the early abandoner, e.g. Germany, and The Netherlands (countries in red color in the world map 2.1).

Germany: In 2005 German election, two voters filed a case in the German Constitutional Court (Bundesverfassungsgericht) because their appeal to scrutinize the election was not heeded by the Committee. They argued that using electronic voting machines to conduct the election was unconstitutional. Furthermore, they added that these machines could be hacked, hence results of the 2005 election could not be trusted on the grounds of public examinability of elections according to German Constitution (Basic Law for the Federal Republic of Germany) [Ger]. The Court noted that, under the constitution, elections are required to be public in nature [Ger]:

> The principle of the public nature of elections requires that all essential steps in the elections are subject to public examinability unless other constitutional interests justify an exception. Particular significance attaches here to the monitoring of the election act and to the ascertainment of the election result.

In its verdict, the court did not rule out or prevent the usage of electronic voting machines, but suggested to make the process more transparent and trustworthy [Ger]:

> The legislature is not prevented from using electronic voting machines in the elections if the constitutionally required possibility of a reliable correctness check is ensured. In particular, voting machines are conceivable in which the votes are recorded elsewhere in addition to electronic storage. This is for instance possible with electronic voting machines which print out a visible paper report of the vote cast for the respective voter, in addition to electronic recording of the vote, which can be checked prior to the final ballot and is then collected to facilitate subsequent checking. Monitoring that is independent of the electronic vote record also remains possible when systems are deployed in which the voter marks a voting slip and the election decision is recorded simultaneously,

or subsequently by electronic means in order to evaluate these by electronic means at the end of the election day.

The Netherlands: The Netherlands was among a few countries who adopted electronic voting in the early nineties (1990), but it did not go very well in the long run and was abolished in 2008 [Jacobs and Pieters, 2009]. The reason for abolishing the electronic voting was that the voting machines used in elections were susceptible to many attacks, and the results of elections conducted using these machines were not publicly verifiable. Besides, a Dutch public foundation, Wij vertrouwen stemcomputers niet (We do not trust voting computers), demonstrated that the e-voting machines used in the election leaks enough information to guess the choice of a voter at a distance of 20 to 30 meters from the polling booths [Net].

Germany and The Netherlands are some of the rare cases where electronic voting was withdrawn because it was not able to replicate the same trust environment as created by paper ballot systems whereas Australia, and India continued with electronic voting despite having the concerns expressed by researchers about the security of system.

India: India, one of the largest democracies in world, uses electronic voting machines (also known as EVMs) for national and state level elections despite the fact that many political parties have raised security concern against them. Moreover, it has already been shown in [Wolchok et al., 2010] that it is possible to manipulate the election results. In their attack, they replaced the parts of electronic voting machine with malicious look alike components. These components were capable of receiving instruction over wireless communication. As a result, any malicious attacker can control these components from nearby vicinity by sending instructions over wireless channel by using a simple hand-held device and can manipulate the results in their favour [1]. India is mainly criticised for keeping the design of electronic voting machines a closely guard secret (security by obscurity). However, it is not impossible to get access of these machines as shown by [Wolchok et al., 2010]. The worst part, the design of these machines were never audited by any independent third party.

Australia: In March, 2015 state election of New South Wales, Australia, the Internet voting system, iVote, was used and 280,000 votes were cast through it. NSW Election commissioner claimed that it was:

[1] https://indiaevm.org/

> It's fully encrypted and safeguarded, it can't be tampered with, and for the first time people can actually after they've voted go into the system and check to see how they voted just to make sure everything was as they intended [NSW].

The voting on iVote opened on Monday March 16 and continued until March 28. On 22 March, two security researchers, Vanessa Teague and J. Alex Halderman, announced that iVote has critical security bug, and they demonstrated that it was good enough to steal any ballot. From their paper [Halderman and Teague, 2015]:

> While the election was going on, we performed an independent, uninvited security analysis of public portions of the iVote system. We discovered critical security flaws that would allow a network-based attacker to perform downgrade-to-export attacks, defeat TLS, and inject malicious code into browsers during voting. We showed that an attacker could exploit these flaws to violate ballot privacy and steal votes. We also identified several methods by which an attacker could defeat the verification mechanisms built into the iVote design.

Basically, New South Wales ran an online election for 6 days on bug-ridden software program which was susceptible to many attacks with a possible outcome of tampered ballot without anyone noticing it.

We do not have to think very hard to figure out the reasons for these debacles. There are various factors for these debacles, but one of the most common denominator among all these debacles, which contributed significantly, is the software/hardware used in the election process had numerous bugs. But this begs the question: why various governments were (Germany, Netherlands)/are (Australia, India) using such poor quality software programs in the first place for conducting elections? In general, no entity related to government or electoral commission develops the software programs for electronic voting, and predominantly it is outsourced to companies having experience in electronic voting software development. Most of these companies produce poor quality software because of unrealistic schedule, lack of proper software testing practices, lack of technical knowledge, etc. In the report, [Lewis et al.] stated in the source of the problem of SwissPost debacle as:

> Nothing in our analysis suggests that this problem was introduced deliberately. It is entirely consistent with a naive implementation of a complex cryptographic protocol by well-intentioned people who lacked a full understanding of its security assumptions and other important details. Of course, if someone did want to introduce an opportunity for manipulation, the best method would be one that could be explained away as an accident if it was found. We simply do not see any evidence either way.

Moreover, more often than not, these software programs are closely guarded secrets and their source code in not open for public scrutiny because of commercial interests of companies [Australian Electoral Commission, 2013] involved in the process. Overall, the whole process lacks transparency, which violates the fundamental principals of democracy, i.e. openness.

The process of turning an idea into a concrete software, also known as software development process, involves requirement gathering, software design, implementation, testing, and maintenance. During this whole process of software development, there are various factors which affect the quality of software. However, throughout this entire software development (and maintenance life) cycle, software testing is the only mechanism for quality assurance, but it is not enough for instilling the confidence in software that it is bug free as stated by Edsger W. Dijkstra [Dijkstra, 1972]:

> Program testing can be used to show the presence of bugs, but never to show their absence!

In the next section, given the mission critical importance of electronic voting software, we will discuss that software testing is not sufficient to achieve the software trustworthiness [Nami and Suryn, 2013], and we should prove the correctness of these software by using formal verification techniques [Beckert et al., 2014]. Furthermore, we will argue that having a formal verification software development methodology [Muñoz et al., 2018] would alleviate the bug problem with two case studies, CompCert [Leroy, 2006] and Athelon, as a supporting evidence. The success of these case studies should be a good motivation for us to adopt formal method for electronic voting software development.

2.2 Correctness: Formal Method Approach

Formal verification has been successfully applied in many areas, and some of the notable software programs are verified C compiler CompCert [Leroy, 2006], verified ML compiler CakeML [Kumar et al., 2014], verified LLVM Vellvm [Zhao and Zdancewic, 2012], verified cryptography Fiat-crypto [Erbsen et al., 2019], verified operating system CertiKOS [Gu et al., 2011] and SeL4 [Klein et al., 2009], verified theorem prover Milwa [Myreen and Davis, 2014], verified crash resistant file system FSCQ [Chen et al., 2015], verified distributed system Verdi [Wilcox et al., 2015], mechanisation of Four Color Theorem [Gonthier, 2008], Fundamental Theorem of Algebra [Geuvers et al., 2002], and Kepler Conjecture [Hales et al., 2015]. None of these are toy projects, and it has taken years to develop and verify them. Also, some of these products are used commercially, e.g CompCert is used by the AIRBUS and the MTU (Motoren und Turbinen Union) Friedrichshafen[2], Fiat-crypto is used in the Google's BoringSSL library for elliptic-curve arithmetic[3].

Based on the cost and efforts of these projects, the very basic question to ponder about using formal method to develop software: does formal verification produce bug free software? We give two anecdotes to answer this question. One of the most basic way to break the software is generating random tests and throwing it to the software under consideration [Miller et al., 1990]. [Yang et al., 2011] developed random C program generator and used these programs to test various compilers. In three years of its usage, they have found 325 unknown bugs in various compiler including GCC[4] and LLVM[5]; however, they could not find any bug in the verified component of CompCert. In their own words [Yang et al., 2011]:

> The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users.

[2] https://www.absint.com/compcert/

[3] https://deepspec.org/entry/Project/Cryptography

[4] https://embed.cs.utah.edu/csmith/gcc-bugs.html

[5] https://embed.cs.utah.edu/csmith/llvm-bugs.html

Formal verification is not only helpful in proving the correctness, but sometimes, it helps in uncovering the bugs in design of software. ACL2, a Lisp based theorem prover, helped AMD to uncover a floating point bug in Athelon processor which has survived 80 million floating point test cases! In the paper, Milestones from the Pure Lisp theorem prover to ACL2 [Moore, 2019], Moore, one of the developer of ACL2, writes:

> When AMD developed their translator from their register-transfer language (in which designs are expressed) to ACL2 functions they ran 80 million floating point test cases through the ACL2 model of Athlon's FMUL and their own RTL simulator. However, the subsequent proof attempt exposed bugs not covered by the test suite. These bugs were fixed before the Athlon was fabricated.

There are numerous instances where formal verification was very useful, and it caught the lurking bugs in design in early stage which could never have been found by testing. For electronic voting software used in democratic election, where we can not afford to lose a single ballot or miscalculation or any undefined behaviour, should be developed using formal method techniques. In order to ascertain that the formal verification of voting software has been carried out diligently, one therefore needs to

1. read, understand, and validate the formal specification: is it error free, and does it indeed reflect the intended functionality?

2. scrutinize the formal correctness proof: has the verification been carried out with due diligence, is the proof complete or does it rely on other assumptions?

The above mentioned requirements can be met by publishing or open sourcing both the specification and the correctness proof so that the specification can be analysed, and the proof can be replayed (inside the tool used for verification) by any independent third party. Both need a considerable amount of expertise, but it can be carried out by (ideally more than one group of) domain experts.

2.3 Verifiability: Trust in Electronic Voting

Given that elections are the cornerstone of our democracy, electronic voting software programs should be considered as mission-critical systems, and therefore they should be developed with highest possible rigour. Formal verification is useful in producing the bug free code, but we solely can not establish the trust in the system based on argument of formal verification. The reason is that how would a voter:

- ascertain that it was indeed the verified program that was executed in order to obtain the claimed results?

- ensure that the computing equipment on which the (verified) program is executed has not been tampered with or is otherwise compromised?

Recall that the notion of (end-to-end) verifiability in electronic voting is ascertaining the outcome of an election without trusting any machine involved in the process. In general, formal verification is certainly a necessary thing in developing the software programs for electronic voting, but it is not sufficient because it does not provide verifiability. Combing both *verification* of the software program that counts votes, and *verifiability* of individual counts are critical for building trust in an election process. These two facts, *verification* and *verifiability*, can also be viewed as the two sides of a coin from the perspective of two major stake holders of a democracy: i) electoral commission/government, and, ii) voters/participants. Using a formally verified software program to count the ballots would increase the confidence of an electoral commission that it has announced and published the correct result. Moreover, the published result always verifies, which boost the confidence of voters in the system.

Given the mission-critical importance of correctness of vote-counting, both for the legal integrity of the process and for building public trust, it is crucial to replace the currently used black-box software for vote-counting with a counterpart that is both verified and produces evidence which can later be used to certify the outcome of election [Bernhard et al., 2017] [Rivest, 2008].

2.3.1 Scrutiny Sheet

A scrutiny sheet is the tabulation of relevant data to verify the result of election. The idea of requiring that computations provide not only results but also a witness attesting to the correctness of the computation is not new and has been put forward in [Sullivan and Masson, 1990] [McConnell et al., 2011] [Arkoudas and Rinard, 2005], and, in the context of electronic voting by [Schürmann, 2009] [Pattinson and Schürmann, 2015]. In general, the idea of computation is that a computable function f takes an input x and produces output y (figure 2.2); however, in case of certified computation, the computable function f on the given input x, not only produces the output y, but it also produces a witness w for the fact that $f(x) = y$ (Figure 2.3).

Figure 2.2: Function f computing y on input x

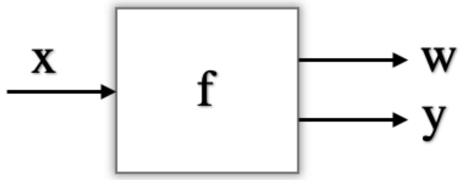

Figure 2.3: Function f computing y and producing witness w on input x

As a simple example, below is a certificate which has been generated by a program which computes the greatest common divisor of two numbers. The program has produced the certificate (piece of data) on the concrete input 34 and 21:

```
gcd 34 21
-------------
gcd 21 13
-------------
```

```
     .
-------------
gcd 1 0
-------------
   1
```

In order to verify the correctness of the computation of greatest common divisor, we need to make sure that one of the rules of Euclidean algorithm, given below, is applicable at each line of the certificate:

1. Rule-zero: ∀ x, gcd x 0 = x
2. Rule-inductive: ∀ x y, gcd x y = gcd y (mod x y)

The same certificate (augmented with rules and variables instantiated) from certificate-checker perspective.

```
gcd 34 21 = gcd 21 13 Rule-inductive: x := 34, y := 21, mod 34 21 = 13
--------------------------------------------------------------------
gcd 21 13  = gcd 13 8 Rule-inductive: x := 21, y := 13, mod 21 13 = 8
--------------------------------------------------------------------
     .
--------------------------------------------------------------------
gcd 1 0 = 1   Rule-zero: x := 0
```

Unfortunately, the example we gave, greatest common divisor, is very simple and not very helpful to put forward the usefulness of certificate in perspective. However, this approach, generating a certificate to certify the computation later, is very useful in context of complicated and unverified programs. One such example is algorithmic library LEDA [Mehlhorn and Näher, 1995] (Library of Efficient Data Types and Algorithms) written in C++. Checkers are an integral part of the LEDA which can later be invoked by user to certify that the result produced by unverified code is correct. Initially, the checkers came with library were unverified, and Kurt Mehlhorn defended this decision by admitting that [Alkassar et al., 2014]:

> Checkers are simple programs with little algorithmic complexity. Hence, one may assume that their implementations are correct.

Later, the checkers [Alkassar et al., 2014] were verified by using VCC [Cohen et al., 2009], and Isabelle/HOL [Nipkow et al., 2002b]. One advantage of this approach is that it is easier to formally verify the checker than the algorithm itself because checkers are very simple in nature, and this approach scales very well.

One may ask the question that can we follow the same approach for vote counting, i.e. unverified counting code, and verified checker? The answer is: it depends. If the verified checker validates the result, i.e. it returns true on the certificate generated by an unverified vote counting program, everything is fine from every stakeholders' perspective. However, what about the situation when the verified checker invalidates the result, i.e. it returns false on the certificate generated by the unverified vote counting program? This kind of situation must be dealt carefully by an electoral commission, and the commission should inspect everything carefully including the vote counting software and various other thing involved in the process. This inspection would definitely be time consuming leading to delay in the result declaration, cost money leading to increase in the election budget, but more importantly, it would hamper the confidence of voters, adding more scepticism to public opinion towards the electronic voting [Avgerou et al., 2019]. To eliminate this kind of problematic situation, we propose: i) a formally verified vote counting software which produces the result with evidence (certificate), and ii) formally verified certificate checker. The advantage of this approach is that the result produced is always correct (modulo specification), a confidence building measure for electoral commission. Furthermore, the verified checker would always return true on the evidence (certificate) produced by the verified vote counting program, confidence booster for voters into the deployed system.

2.4 Privacy

So far we have argued that given the importance of vote counting software in an election, we should develop it with highest possible rigour, i.e. formal verification and open source it for public scrutiny. Furthermore, to increase the public trust in vote counting process, we should strive for verifiability by tabulating all the relevant data (scrutiny sheet) on a public bulletin board. Many jurisdictions follow some derived version of these practises, e.g. ACT

Electoral Commission[6] has published all the ballots and vote counting software since 2001, the year electronic voting was introduced (surprisingly, this year, 2020, ACT Electoral Commission did not publish the vote counting software, a step back than forward). However, for preferential voting schemes publishing all the ballots in plaintext could lead to ballot identification via Italian attack [Otten, 2003], which we have already discussed in the previous chapter (see 1.5). This may appear surprising in the first sight, but if we analyse it more carefully, we can see that verifiability requirement is causing this privacy issue. Indeed, privacy and verifiability are conflicting requirements [Jonker et al., 2013]. On the one hand, (end-to-end) verifiability, in a broad sense, is about making sure that every voter can see that her ballot is included in the final tally and the final tally is produced based on all the published ballots, while on the other, privacy is about not letting any one link a ballot to a particular voter, even if that particular voter wants. The only way to resolve the tension between privacy and verifiability is to use various cryptographic primitives, which hide the data by using encryption (privacy) and ensure that every claim is accompanied by a mathematical proofs (verifiability) (these proofs are in form of data which can be checked by anyone and commonly known as zero-knowledge-proof).

The Italian attack may seem far-fetched to the reader; however, a political scientist, Dr. Kevin Bonham, was able to link 15 ballots, when he was studying the Tasmania Senate election, to its voters by looking at the preferences [Bonham]. The preferences on these 15 ballots were so rare, amongst all the cast ballots, that he conjectured that these votes could be a recommendation from some lobby group or some technological issue. Later, he found the voters of these 15 ballots on a private Facebook group where one person came up with a rough order, which eventually led to this particular order after some refinement. He pointed [Bonham] towards a potential privacy and coercions issue:

> So in theory a coercive person could direct a voter to vote in a certain way, and then check the files to verify whether anyone in a booth, or at all, had voted in that way. The coercer could even choose to "sign" the directed vote with an unlikely combination of candidates or an unlikely set of repetitions, not affecting the fate of the vote, to make it more likely that the coerced vote would be unique in the whole state and not just at booth level. That way if

[6] https://www.elections.act.gov.au/elections_and_voting/electronic_voting_and_counting

the vote did not appear, they would know the voter had not voted as they directed.

We champion, considering the importance of privacy in elections, that cryptographic methods should be used to hide (encrypt) the preferences in the ballot of a given voter. Moreover, every claim about the (encrypted) ballot should be accompanied by mathematical proofs (in form of data) so that the claims can be verified by any independent third party, including the voter itself. At this point, we would like to point that cryptography makes the auditing difficult for general public because of the complex mathematics involved in the process. We believe nonetheless that there would enough voters and independent auditors, having the knowledge of required cryptographic schemes, who would be able to audit the election and ascertain the outcome.

2.5 Summary

In this chapter, we argued that to make the electronic voting more trustworthy and privacy preserving, we need three ingredient:

1. correctness: formal verification of vote counting software
2. verifiability: tabulation of all the relevant data on a public bulletin board
3. privacy: hiding the options of a voter by means of cryptography

The system, by combing these three concepts, would be a formally verified (encrypted) vote counting software that not only computes the final result but additionally produces an independently verifiable certificate, which attests to the correctness of the computation. The major advantage of a certificate-producing formally verified vote-counting program, all the external parties or stakeholders can satisfy themselves to the correctness of the count by checking the certificate. Moreover, we also argued that including a formally verified certificate checker would boost the confidence of both, electoral commission and voters.

In the next chapter, we will briefly discuss about the Coq theorem prover and basic cryptographic primitives which would enable us later in counting encrypted ballots (without revealing any information about the voters' choice).

Chapter 3

Theorem Prover and Cryptography

> All our knowledge begins with the senses, proceeds then to the understanding, and ends with reason. There is nothing higher than reason.
>
> *Immanuel Kant*

A proof assistant or theorem prover is a computer program which assists users in development of mathematical proofs. Basically, the idea of developing mathematical proofs using computer goes back to Automath (automating mathematics) [de Bruijn, 1983] and LCF [Milner, 1972]. The Automath project, 1967 until the early 80's, was initiative of De Bruijn, and the aim of the project was to develop a language for expressing mathematical theories which can be verified by aid of computer. Moreover, the Automath was first practical project to exploit the Curry-Howard isomorphism (proofs-as-programs and formulas-as-types). DeBruijn was likely unaware of this correspondence, and he almost re-invented it. The Automath project can be seen as the precursor of proof assistants NuPrl [Constable et al., 1986] and Coq [Bertot et al., 2004]. Some other notable proof assistants are Nqthm/ACl2 [Kaufmann and Strother Moore, 1996], PVS [Owre et al., 1992], HOL (a family of tools derived from LCF theorem prover) [Slind and Norrish, 2008] [Harrison, 1996] [Nipkow et al., 2002a], Agda [Norell, 2009], and Lean [de Moura et al., 2015].

Chapter overview: This chapter is an overview of the Coq theorem prover and cryptographic primitives. In section 3.1, we give a brief overview of the theoretical foundation, calculus of construction and calculus of inductive construction, of Coq. In section 3.1.3, we discuss the difference between

Type and *Prop*, which is very crucial from program extraction point of view (the goal of our formalization is not only proving the correctness of Schulze method but extracting OCaml/Haskell code to count ballots). In section 3.1.4, we focus on dependent type and how it leads to correct by construction paradigm by demonstrating a type safe printf function. Section 3.1.5 focuses on Coq specification language *Gallina* with an example demonstrating that why writing proofs using Gallina is difficult and cumbersome, and how it can be eased by using tactics. Finally, in section 3.1.6, we take philosophical route to justify that why we should trust in Coq proofs, even though they do not appear anywhere near to a proof written by a human.

In section 3.2, we give some historical context and modern day usage of cryptography. In the following section 3.2.1, we describe *Group* which is the underlying algebraic structure for the Diffie-Hellman construction (3.2.2). In the next two sections, we describe the ElGamal encryption (3.2.3) and Homomorphic Encryption (3.2.4). In addition, we show the both, multiplicative and additive, homomorphic property of the ElGamal encryption. We explain the concept of zero-knowledge proof and zero-knowledge proof of knowledge in Section 3.2.5. In the next two sections, we discuss sigma protocols (3.2.6), an efficient way to achieve zero-knowledge proof, and commitment schemes (3.2.7), a cryptographic protocol to force two mutually distrusting parties to behave honestly with the explanation of Pedersen commitment scheme based on discrete logarithm. Finally, we give a brief summary pointing to the resources for theorem proving and cryptography.

3.1 Coq: Interactive Proof Assistant

Coq is an interactive proof assistant (theorem prover) based on the theory of calculus of inductive construction (CIC) [Paulin-Mohring, 1993] which itself is an augmentation of the calculus of construction (CoC) [Coquand and Huet, 1988]. The underlying theory of calculus of construction and calculus of inductive construction is (typed) lambda calculus, so before we describe the calculus of construction and calculus of inductive construction syntax and its typing judgement, we will take a brief detour to explain the different variants of lambda calculus starting from untyped lambda calculus and moving up progressively by adding various abstractions. Later, we will show that these all variants, including calculus of construction, can be abstracted into one framework, pure type system (PTS) [Berardi, 1988] [Barendregt, 1992]. In addition, pure type system can be extended with three rules, inductive data type, pattern matching, and recursion to accommodate calculus of inductive construction.

Lambda calculus was invented by Alonzo Church in the 1930s, and his motive was to use lambda calculus as a foundation for formal mathematics, specifically the notion of computable function by means of an algorithm. It is a simplest programming language having just three constructs, i.e. variable, application, and abstraction, and the abstract syntax tree of lambda calculus is:

T = V (* Variables *)
| λ V. T (* Abstraction *)
| T T (* Application *)

Using these three rules, we can construct the lambda terms corresponding to various mathematical notions. For example, we can represent *True* as $\lambda x.\lambda y.x$, *False* as $\lambda x.\lambda y.y$, *Zero* as $\lambda f.\lambda x.x$, *One* as $\lambda f.\lambda x.fx$, etc. However, there is nothing which is stopping us from constructing a lambda term which has no apparent meaning, e.g. applying a variable x to itself leading to a lambda term xx. To avoid situation like this, we extend the lambda calculus with another abstraction called *type*. Moreover, we add *typing judgement* (rule) that dictates which term is well-typed and which one is not. This new lambda calculus, augmented with type, is known as *Simple Typed Lambda Calculus*, represented as $\lambda^{\rightarrow}$. The abstract syntax tree for simple typed lambda calculus is:

$\mathcal{T} = \mathcal{V}$ (* Type Variable *)
| $\mathcal{T} \rightarrow \mathcal{T}$ (* Arrow Type *)

T = V (* Variables *)
| λ V : $\mathcal{T}$. T (* Abstraction *)
| T T (* Application *)

The typing judgement is a relation between *type* and *term* in some abstract typing context Γ. The Γ is a set or list of typing assumption of the form $x : A$, meaning term x is of type A. Moreover, $\Gamma \vdash x : A$ means that term x has type A in the context Γ. The typing judgement of *Simple Typed Lambda Calculus* has three rules, *Var, Abstraction*, and *Application*, to ensure that the terms are well-typed:

- Var:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

- Abstraction:

$$\frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash (\lambda x : A.e) : A \to B}$$

- Application:

$$\frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash x : A}{\Gamma \vdash f\ x : B}$$

Now these three typing judgements reject the term xx because it is not well-typed term, and this can be inferred from the Application rule. The Application rule states that for $f\ x$ to be a well typed term of some type B, the f must have a type $A \to B$ for some type A and x must have the type A. Following the Application rule, for xx to be well typed, the x must have an arrow type $A \to B$ and type A simultaneously in some typing context Γ. However, it is not possible that $A = A \to B$ leading to rejection of the term xx.

Simple typed lambda calculus is great for many practical purposes, except it is verbose. Consider a function which takes an input and simply returns it, also known as identity function. If we extend the type variable set $\mathcal{V}$ with two base type, *nat* for the type of natural numbers and *bool* for type of boolean values, we can represent an identity function on boolean value as $\lambda x : bool.x$ and on natural number as $\lambda x : nat.x$. In general, we would have one identity function per type. We can abstract these types into a type variable, but we need to type these type variables as well to keep everything well typed. Consequently, abstracting the types over type variable, which itself is of sort *kinds* and represented as $*$, leads to *Second Order Lambda Calculus* ($\lambda 2$), and now the identity function over different types can be abstracted into a single function: $\lambda \alpha : \star.\lambda x : \alpha.x$. There are various other variants or abstractions of typed lambda calculus, which we would not discuss here, that can be categorized into:

- Terms depending on terms ($\lambda^{\to}$)
- Terms depending on types ($\lambda 2$)
- Types depending on types ($\lambda \underline{\omega}$)
- Types depending on terms (λP)

All these variations of the lambda calculus can be captured into an unified framework known as *Pure Type System* [Berardi, 1988] [Barendregt, 1992]. Unlike the simple typed lambda calculus ($\lambda^{\rightarrow}$) where terms and types live in two disjoint worlds, pure type system blurs this distinction between types and terms and allows the dependencies between them. The abstract syntax of pure type system:

T = V (* *variable* *)
| C (* *constant* *)
| T T (* *application* *)
| λ V : T. T (* *abstraction* *)
| $\prod$ V : T. T (* *dependent function type* *)

The pure type system is parametrized by three specifications: i) set of sorts S, ii) set of axioms A, and iii) set of rules R such that:

- S is a subset of C, i.e. $S \subseteq C$.
- A is the set of axioms of form $c : s$ where $c \in C$ and $s \in S$, i.e. $A \subseteq C \times S$.
- R is the set of rules of form (s_1, s_2, s_3) such that s_1, s_2, and $s_3 \in S$, i.e. $R \subseteq S \times S \times S$.

The typing judgement for the pure type system, in a typing context Γ, is defined by following rules (s ranges of S, and x ranges over V with usual notion of variable capture avoidance):

- Axiom:
$$\frac{c : s \in A}{\Gamma \vdash c : s}$$
- Start:
$$\frac{\Gamma \vdash A : s}{\Gamma, x : A \vdash x : A}$$
- Weakening:
$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash C : s}{\Gamma, x : C \vdash A : B}$$
- Product:
$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B : s_2}{\Gamma \vdash (\prod x : A.\, B) : s_3}$$

- Application:

$$\frac{\Gamma \vdash F : (\prod x : A.\, B) \quad \Gamma \vdash a : A}{\Gamma \vdash F\, a : B\, [x := a]}$$

- Abstraction:

$$\frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\prod x : A.\, B) : s}{\Gamma \vdash (\lambda x : A.\, b) : (\prod x : A.\, B)}$$

- Conversion:

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : s \quad B =_{\beta} B'}{\Gamma \vdash A : B'}$$

3.1.1 Calculus of Construction/Inductive Construction

The calculus of construction is a higher order natural deduction style proof system for constructive proofs where every proof a typed λ-abstractions. Using the pure type system syntax, it can be expressed as:

$S =$

$$\{Prop\} \cup \{Type_i \mid i \in \mathbb{N}\}$$

$A =$

$$\{Prop : Type_0\} \cup \{Type_i : Type_{i+1} \mid i \in \mathbb{N}\}$$

$R =$

$$\left\{ \begin{array}{l} (Prop, Type_i, Type_i)\ i \in \mathbb{N} \\ (s, Prop, Prop)\ s \in S \\ (Type_i, Type_j, Type_{max(i,j)}) \end{array} \right\}$$

The sort *Prop* captures the type of expressions which represent logical proposition, while the sort *Type* captures the computational content. The calculus of construction is powerful enough to encode inductive definitions [Pfenning and Paulin-Mohring, 1989], but one of the main drawback is efficiency of computation over these encoded inductive definitions. Moreover, some other properties could not be proven [Geuvers, 2001]. Therefore, [Paulin-Mohring, 1993] introduced Inductive definitions, pattern matching, and fixpoint in the calculus of construction to make the data structure representation more efficient. Below is the (incomplete) syntax of calculus of inductive construction:

T = ... (* Pure Type System *)
| Ind { V : T := **V : T**}.V (* inductive definition)

| case T of **V => T** (* pattern matching *)
| fix$_n${ V : T := T } (* recursion *)

3.1.2 Inductive Type

As we mentioned above that inductive types are basic building block for encoding various data structures in the Coq (calculus of inductive construction). The keyword to declare an inductive data type in Coq is *Inductive*. For example, a length index list whose elements belong to a type A can defined as (also known as vector):

```
Inductive Vector (A : Type) : nat -> Type :=
| Nil : Vector A 0
| Cons n : A -> Vector A n -> Vector A (S n).
```

Now we can define various functions for the vector data structure. For example, we can define a function to append two vectors for length n and p as:

```
Fixpoint append {A n p} (v : Vector A n) (w : Vector A p)
  : Vector A (n + p) :=
  match v with
  | Nil _ => w
  | Cons _ _ a v' => Cons _ _ a (append v' w)
  end.
```

The expressiveness of Coq allows to encode various correctness properties at type level. In our example of append, the correctness criteria states that appending a vector of length n with a vector of length p yields a vector of length $(n + p)$. In other words, the function append is "correct-by-construction". During our formalisation, we have encoded our vote counting as an inductive data type with various assertions (correctness specification) appearing at type level. These assertions at the type level enforce that only a "correct" term of the vote counting inductive data type can be constructed (*correct-by-construction*).

We would like to point that the current underlying theory of Coq has been extended with co-inductive types [Giménez, 1995]; however, the discussion of co-inductive types is not very relevant for this thesis.

3.1.3 Type vs. Prop: Code Extraction

Every term in Coq has a type, and the term could be either a logical proposition or a computational term. The type of logical proposition is *Prop*, while the type of computational term is *Type*. This distinction between the type of logical proposition (Prop) and the type of computational term (Type) provides a mechanism to extract functional program directly from Coq proof script. During the extraction process [Letouzey, 2008], every term of type Prop is erased and no longer exists in the extracted code, and only the terms of type Type are translated into the target language (OCaml/Haskell/Scheme). Because of this, Coq in general does not allow the case analysis on the terms (logical objects) of type Prop when the goal is not in Prop, but in certain cases it can be achieved (we call this special case reification and explain next).

3.1.3.1 Reification

Sometimes it is very natural to express certain properties or definitions in the Prop than in the Type. Moreover, the definitions/terms in the Prop are self contained and very intuitive for human understanding. The only problem is that the terms of the type Prop do not carry any computational content but only the proof part. However, we can escape this situation if the term of type Prop is decidable predicate (boolean predicate) and its domain is finite. In the case of decidable predicate in Prop over a finite domain, we can extract a witness constructively by enumerating the elements of the finite domain in a list and using a linear search program that tries the decidable predicate on every element of the enumerated list.

Figure 3.1 is a reification Coq code which produces a Type level witness, *existsT*, from a Prop level witness, *exists*, by iterating through all the elements of finite type A (the finiteness of A is captured by the list l) [Firsov and Uustalu, 2015a]. The program is written in the proof mode, hence the clutter of tactics, but the basic idea is that we inspect every single element of A, represented as list l, and look for the element which satisfies the Prop level assertion, (exists x, In x l $\wedge$ P x = true), and this element is precisely our witness which is wrapped in a type level existential, existsT x, P x = true. Another way to look at the reification is that the Prop level assertion, (exists x, In x l $\wedge$ P x = true), merely postulate the existence of an element without telling which one, and we find this element by looking over all the elements of finite type A, as we can see it is pretty evident from the extract OCaml code 3.2 from the Coq proof 3.1 (the proofs terms are erased and only computational terms contribute to the extracted function).

We have used many standard tricks like this to make our formalization more accessible for human inspection. For example, we have two definitions, one in Prop and other in Type, of winner, loser, and path. The rationale behind two definitions for the same thing is that Prop definition is very natural and easy to understand compared to their Type counter part. Furthermore, we have shown that they are equivalent to each other and used the definitions in Type for computation. The biggest advantage of this is that anyone can understand our formalisation by just looking at the Prop definitions, without looking at the complicated Type definitions, because they are self contained. Also, there is a nice Coq library ConstructiveEpsilon[1] which uses the similar trick as ours; however, we have not used this library in our formalization.

3.1.4 Correct by Construction: Type Safe Printf

One of the highly sought feature of Coq is dependent type, a type which is parametrised by value. The expressiveness of dependent type makes it possible to express specification at type level, and these specifications enable larger set of logical errors to eliminate at compile time.

The printf in the C programming language is vararg (variable length argument) function, and it simply trusts the format string to accurately describe the arguments. However, sometimes this trust can be exploited during the execution of a program by deliberately making sure that the format string does not describe the arguments accurately. The type system of C programming language is not strong enough to forbid it during the compile time, but if a language, Coq in our case, has dependent type, this exploit can be averted during compile time. Therefore, using the expressiveness of dependent type, we can construct a type-safe version of printf [Pierce, 2004], which is not amenable to format string exploit.

Our goal is to define a type safe printf function which generates com-

[1] https://coq.github.io/doc/master/stdlib/Coq.Logic.ConstructiveEpsilon.html

```
Require Import Coq.Lists.List.
Import ListNotations.

(* type level existential quantifier *)
Notation "'existsT' x .. y , p" :=
  (sigT (fun x => .. (sigT (fun y => p)) ..))
    (at level 200, x binder, right associativity,
     format "'[' 'existsT'  '/  ' x  ..  y ,  '/  ' p ']'")
  : type_scope.

(* the following shows that a decidable (or boolean valued)
   predicate on a finite list
   can always be reified in terms of strong existence *)
Theorem reify {A: Type} (P: A -> bool) : forall (l: list A),
    (exists x, In x l /\ P x = true) -> existsT x, P x = true.
Proof.
  refine (
      fix Fn l :=
        match l with
        | [] => fun H => _
        | h :: tl => fun H => _
        end).
  contradict H. intro.
  destruct H as [x [H1 H2]].
  firstorder.

  assert (Hbiv: {P h  = true} + {P h <> true}).
  decide equality.
  destruct Hbiv as [Htrue | Hfalse].
  exists h. assumption.
  specialize (Fn tl). apply Fn.
  destruct H as [x [H1 H2]].
  destruct H1. subst.
  firstorder. exists x.
  firstorder.
Defined.
```

Figure 3.1: Coq Code for Reification

```
(** val reify : ('a1 -> bool) -> 'a1 list -> ('a1, __) sigT **)

let rec reify p = function
| Nil -> assert false (* absurd case *)
| Cons (h, tl) ->
  let hbiv = match p h with
             | True -> Left
             | False -> Right in
  (match hbiv with
   | Left -> ExistT (h, __)
   | Right -> reify p tl)
```

Figure 3.2: Extracted OCaml Code from the Coq Code

piler error when the given format string does not describe the arguments accurately. For example, *type-safe-printf "%d %s" "hello Coq" 42* should be a compiler error because %d is a directive for integer value, but the type of argument, "hello Coq", is string. In addition, type-safe-printf should print the arguments when the format string describes the arguments accurately. For example, *type-safe-printf printf "%s %d" "hello Coq" 42* should print the string "hello Coq 42" because the first directive of format string, %s, and type of argument, "hello Coq", are aligned. Similarly, the second directive of format string, %d, is also aligned with the type of argument, 42.

The high level idea is that *type-safe-printf* should return a type which is solely constructed based on the *format-string*, and this return type should unify with the arguments given to the type-safe-printf. For example, the return type of *type-safe-printf "%s %d"* should be *String -> Integer -> String* because %s is directive for string and %d is directive for integer. Assuming that our type-safe-printf behaves in this way, then *type-safe-printf "%s %d" 42 "hello Coq"* would be a complier error because the type of the first argument, 42, is integer, while *type-safe-printf "%s %d"* expects a string.

Now getting into the details, the idea is to split the type-safe-printf arguments into two parts: i) format string, and ii) arguments (values to be printed). For example, printf "%s %d" "hello Coq" 42 would be split into "%s %d", and "hello Coq" 42. Based on the format string, we design two functions: i) a type level function, and ii) a value level function. The type level function would take a format string and returns a variadic function type, e.g. on a format string "%s %d", it would return a function with type signature *string -> Integer -> string*. The value level function, whose type signature is constructed by the type level function, would take the arguments, values to be printed, as input. If the type of arguments is aligned with the type constructed by the type level function on the format string, we proceed to print the string, otherwise we generate compiler error.

To accomplish the functionality of type-safe-function, we defined an abstract syntax tree, *format*, to make explicit the characters we are interested in format string. In our case, it is integer (Fint), string (Fstring), or any other character (Fother) (Fend represents an empty string and is there to hint the end of a format string).

```
(* abstract syntax tree *)
Inductive format :=
| Fend : format
| Fint : format -> format
| Fstring : format -> format
| Fother : ascii -> format -> format.
```

Now we define a function, *format_string*, that takes a string, which represents a format string, and returns a *format* data type. For example, *format_string* on the input "%s %d" would return Fstring (Fother ' ' (* space character *) (Fint Fend)).

```
(* turn the format string into abstract syntax tree *)
Fixpoint format_string (inp : string) : format :=
  match inp with
  | EmptyString => Fend
  | String ("%"%char) (String ("d"%char) rest) =>
       Fint (format_string rest)
  | String ("%"%char) (String ("s"%char) rest) =>
      Fstring (format_string rest)
  | String c rest => Fother c (format_string rest)
  end.

Eval compute in format_string "%s %d".
  = Fstring (Fother " " (Fint Fend))
     : format
```

We will use the *format* data type as a hinge to construct a function, *interp_format*, that returns variadic function in type scope (type level) and a function, *interp_value*, that returns variadic function in value scope (value level). Moreover, the type of value level function exactly matches or unifies with the type returned by *interp_format* (in fact, the return type of *interp_value* is constructed by *interp_format*). This is key step where we hook a value level function with a type level function to make sure that format string and arguments align with each other. One of the striking feature of this example is to use a value (format string) to construct two things, a type level function and a value level function, and glue them together to get a better security.

```
(* construct the type level function from abstract syntax tree *)
Fixpoint interp_format (f : format) : Type :=
  match f with
  | Fint f => Z  -> interp_format f
  | Fstring f => string -> interp_format f
  | Fother c f => interp_format f
  | Fend => string
  end.


Eval compute in interp_format (Fstring (Fother " " (Fint Fend))).
  = string -> Z -> string
     : Type

(* value level function whose type is constructed
    on fly by interp_format function *)
Fixpoint interp_value (f : format) (acc : string) :
  interp_format f :=
  match f with
  | Fint f' => fun i => interp_value f' (acc ++ of_Z i)
  | Fstring f' => fun i => interp_value f' (acc ++ i)
  | Fother c f' => interp_value f' (acc ++ String c EmptyString)
  | Fend => acc
  end.
```

Finally, we glue all these functions together to define the type-safe-printf function. In addition, we evaluate it on two inputs: i) type-safe-printf "%d %s" "hello Coq" 42, and ii) type-safe-printf "%d %s" 42 "hello Coq".

```
Definition type-safe-printf s := interp_value (format_string s) "".
```

```
Eval compute in  type-safe-printf "\%d \%s" "hello Coq"%string 42.
(* Error: The term ""hello Coq"%string" has type "string"
while it is expected to have type "Z". *)

Eval compute in  type-safe-printf "\%d \%s" 42 "hello Coq"%string.
(*  "\0b101010 \hello Coq"%string. The number
42 is printed in binary *)
```

3.1.5 Gallina: The Specification Language

The example, type safe printf function, I gave in the previous section was encoded in Coq's specification language Gallina. Gallina is a highly expressive specification language for development of mathematical theories and proving the theorems about these theories; however, writing proofs in Gallina is very tedious and cumbersome. Furthermore, it is not suitable for large proof development. In order to ease the proof development, Coq also provides tactics. The user interacting with Coq theorem prover applies these tactics to build the Gallina term, which otherwise would be very laborious.

To demonstrate our point, we have written two proofs that addition on natural number is commutative. First proof, *addition_commutative_gallina*, is written using Gallina, while the second proof, *addition_commutative_tactics*, is written using the tactics. In general, we write programs directly in Gallina and use tactics to prove properties about the programs. However, there is no fixed set of rules, and tactics can be used to write programs with dependent types (which we have done during this formalization).

```
(* proof written using the combination of Gallina  terms
   and tactics. The reason for using tactics is that
   in the inductive case, terms are complicated
   and difficult to handle  *)
Lemma add : forall n m : nat, n + m = m + n.
Proof.
  refine (fix Fn n :=
            match n as n' return (n = n' -> forall m, n' + m = m + n') with
            | 0 =>
              fun H => fix Fm m : 0 + m = m + 0 :=
                match m as mt return 0 + mt = mt + 0 with
                | 0 => eq_refl
```

```
                    | S m' => eq_ind (0 + m')
                                   (fun t : nat => S m' = S t)
                                   eq_refl
                                   (m' + 0)
                                   (Fm m')
                    end
                 | S n' =>
                   fun H => fix Fm m : S n' + m = m + S n' :=
                      match m as mt return S n' + mt = mt + S n' with
                      | 0 =>  eq_ind_r (fun t : nat => S t = S n')
                                  eq_refl (Fn n' 0)
                      | S m' => _
                      end
                end eq_refl).
  simpl. rewrite <- (Fm m').
  rewrite (Fn n' (S m')).
  simpl.
  repeat apply f_equal.
  rewrite (Fn n' m').
  exact eq_refl.
Qed.


(* proof written using tactics *)
Lemma addition_commutative_tactics :
  forall (n m : nat), n + m = m + n.
  intros n m; try omega.
Qed.
```

3.1.6 Trusting Coq proofs

In general, Coq proofs are nowhere similar to a mathematical proof written by trained mathematician. Also, these proofs are verbose and fairly long, so a very fundamental question is: why should we accept or believe in a proof written in Coq [Pollack, 1998]? Generally, the answer of accepting or trusting Coq proofs is two-fold: i) is the logic, calculus of inductive construction, sound?, and ii) is the implementation correct? The logic has already been reviewed by many peers and proved correct using some meta-logic, therefore the answer of our question about trusting Coq proof hinges on the implementation. The Coq implementation, written in OCaml, has two parts, the type checker (small kernel), and tactic language to build the proofs. We lay our trust in type checker because it is a small kernel and can be manually inspected. Furthermore, if there is a bug in tactic language, which often is the case, build proof would not pass the type checker. Also, we can use the publicly available proof checkers written by experts and inspected by many others. In addition, to increase the confidence, there have been efforts to certify type checker [Appel et al., 2003] [Barras, 1996], verify meta theory of one proof system in other [Anand and Rahli, 2014], self certificate of theorem prover [Harrison, 2006]. However, no system can prove its own consistency (Gödel's second incompleteness theorem), therefore trusting human judgement is inevitable.

3.2 Cryptography

The word cryptography comes from the two Greek words: *kryptós*, meaning *hidden*, and *gráfein* meaning *to write*. As a matter of fact, in the past, hidden writing (cryptography), using the symbol replacement, has been used to conceal messages. For example, the earliest known usage of cryptography (symbol replacement) goes back to ancient Egyptian (Khnumhotep II, 1500 BCE); however, the purpose of replacing one symbol by other was not to protect any sensitive information but to enhance the linguistic appeal. The first known usage of cryptography to conceal sensitive information goes back Mesopotamians (1500 BCE), where they used it to hide the formula for pottery glaze. Fast forward, around 100 BCE, Julius Caesar wrote a letter to Marcus Cicero using a method, now known as *Caesar cipher*, which would shift each character in letter by 3 position right with wrapping around, i.e. X would wrap A, Y would wrap to B, and Z would wrap to C. Decryption was 3 character left shift. Using the tools of modern mathematics, encryption and decryption in Caesar cipher are modular addition and modular subtraction (modulo 26), respectively. Overall, cryptography is art and science of making thing unintelligible from everyone, except the intended recipient.

The modern day cryptography originated in 1970 with two ingenious ideas, *Data Encryption Standard (DES)* [Standard et al., 1999], and *Diffie-Hellman Algorithm* [Diffie and Hellman, 2006]. Data Encryption Standard, developed at IBM in 1970, is a symmetric key encryption algorithm which uses the same key for encryption and decryption. Since its inception, Data Encryption Standard amassed a bad reputation because of National Security Agency (NSA) involvement; however, it had a practicality issue: key management. If the two parties wanted to communicate securely over an insecure channel using the Data Encryption Standard, they needed to agree on a common key. In order to agree on the common key, they needed a secure channel where they could securely communicate the key. The solution to this problem came from Diffie-Hellman key exchange where two parties can exchange the key securely over insecure channel. Moreover, the advent of Diffie-Hellman key exchange started the whole new area of public key cryptography where encryption and decryption key are different. Although Diffie-Hellman key exchange suffers from man-in-the-middle (MITM) attack if used for keys exchange in its naivety, e.g. Logjam [Adrian et al., 2015], so precautions must be taken when using it for key exchange. In 1985, Tahir ElGamal proposed a new public key encryption (and decryption) scheme [ElGamal, 1985], based on the Diffie-Hellman algorithm, which is still used predominately to secure the electronic transactions of the Internet (in fact, Tahir ElGamal is known as "father of SSL (secure socket layer)").

In this thesis, we are mostly concerned about public key cryptography. The basic working principles of modern day cryptography is based on the mathematical principles, e.g. the underlying mathematical principal of Diffie-Hellman algorithm is hardness of computing discrete logarithms in a finite abelian group (group of prime order). Moreover, it is no longer just used to achieving confidentiality or secrecy, but various other things, e.g. integrity, authentication, non-repudiation, digital signature, digital cash, etc. These cryptographic concepts involve various algebraic structures and algorithms to manipulate the object from the algebraic structures.

Now we describe the workings of Diffie-Hellman [Diffie and Hellman, 2006] algorithm, because all the constructions we have used are based on Diffie-Hellman construction. Before we describe the algorithm, we briefly sketch the algebraic structure Group because it is underlying algebraic structure of Diffie-Hellman construction (typically, the underlying structure is multiplicative group of a finite field). Also, note that our definition is influenced by theorem-provers/type-theory because we have written the type signature of group operator $*$ and inverse operator *inv*.

3.2.1 Group

A group is a set G, with a binary operator $* : G \to G \to G$, identity element e, and inverse operator $inv : G \to G$, denoted as $^{-1}$, such that the following laws hold:

- `Associativity`: $\forall\ a\ b\ c\ \in G,\ a * (b * c) = (a * b) * c$
- `Closure`: $\forall\ a\ b \in G,\ a * b \in G$
- `Inverse Element`: $\forall\ a \in G\ \exists\ a^{-1} \in G$, such that $a * a^{-1} = a^{-1} * a = e$. a^{-1} is called inverse of a ($inv\ a$).
- `Identity`: $\forall\ a \in G, a * e = e * a = a$

Furthermore, if a group is commutative, i.e. $\forall\ a\ b\ \in G,\ a * b = b * a$, we call it abelian group (in honour of Niels Henrik Abel). In addition, a group is *cyclic group* if it can be generated by a single element, also known as generator of group and denoted as g, by repeatedly applying the group operator $*$ to itself. Moreover, a group is *finite cyclic group* if it is cyclic and the cardinality of the underlying set (carrier set) G is finite. The cardinality is also known as order of group.

3.2.2 Diffie-Hellman Construction

Now we explain Diffie-Hellman construction. The construction can be divided into two steps:

1. The two communicating parties, say Alice and Bob, agree with shared public parameters which are finite cyclic group G of order p (p is a large prime) and generator element g.
2. After agreeing with public parameters, Alice and Bob initiates the key exchange protocol (assuming that Alice goes first):
   (a) Alice selects a random number a, where $1 < a < p$, computes g^a ($g * g * g ... * g\ a$ times), and shares g^a with Bob.
   (b) Similarly, Bob selects are random number b, where $1 < b < p$, computes g^b, and shares g^b with Alice.
   (c) Finally, Alice computes the key $(g^b)^a$, and Bob computes the key $(g^a)^b$. A basic algebraic simplification on Alice's key and Bob's key would show that they both have the common key g^{ab}.

During the whole process, Eve, the adversary, would have g^a and g^b, but she can not compute the g^{ab} from these two values assuming that discrete logarithm is hard to compute. There are, off course, other attacks, e.g. man in the middle attack [Menezes et al., 2018], Logjam [Adrian et al., 2015], etc. The security property of Diffie-Hellman construction is formalized using complexity theoretic notions, given below (we would not go into the details of complexity theoretic notions):

DL - Discrete Logarithm problem: An instance of *DL* problem states that given a finite cyclic group G, a generator g of G, and an element y, we need to find an element $x \in G$ such that $g^x = y$ (computing this x is believed to be a hard problem).

CDH - Computational Diffie-Hellman problem: An instance of *CDH* problem states that given a finite cyclic group G, a generator g of G, elements g^a and g^b, we need to find the element g^{ab}.

DDH - Decisional Diffie-Hellman Problem: An instance of *DDH* problem states that given a finite cyclic group G, a generator g of G, elements g^a, g^b, and g^c, we need to determining if c is equal to $a * b$, i.e. $c = a * b$, or not, i.e. $c \neq a * b$.

3.2.3 ElGamal Encryption Scheme

In 1985, Tahir ElGamal [ElGamal, 1985] proposed a new encryption system which was based on Diffie-Hellman algorithm. Tahir ElGamal turned the interactive Diffie-Hellman algorithm into a non-interactive, no need for any active second party, by introducing a randomness. The *ElGamal* scheme has three phases:

1. **Key Generation:** The user, say Alice, first chooses a finite-cyclic group G of order p (p is a large prime) and a group generator g. She randomly selects an element x from $\{1, \ldots, p-1\}$ as a private key, computes her public key $h = g^x$. Subsequently, she publishes the (G, g, p, h) and keeps x private.

2. **Encryption:** If any party, say Bob, wants to send an encrypted message m to Alice, then he would randomly select an element r, where 1 < r < p, computes $c_1 := g^r$ and $c_2 := m * h^r$, and send the pair (c_1, c_2) to Alice.

3. **Decryption:** Upon receiving any pair (c_1, c_2), Alice would compute $c_2 * c_1^{-x}$. A basic simplification of $c_2 * c_1^{-x}$ shows that it recovers the plaintext message. The simplification proceeds by replacing the c_2 with $m * h^r$ and c_1 with g^r in $c_2 * c_1^{-x}$. This substitution leads to $m * h^r * g^{-rx}$ which upon

further simplification by replacing the h with g^x leads to $m * g^{xr} * g^{-rx}$. Using the same base rule, the term $m * g^{xr} * g^{-rx}$ can be written as $m * g^{xr-rx}$. Since $xr = rx$, so we can replace $m * g^{xr-rx}$ with $m * g^0$. The term $g^0 = e$ (the identity of group G) and using the right identity group law, we can replace $m * e$ by m.

3.2.4 Homomorphic Encryption

Homomorphic encryption is an encryption scheme which allows us to perform useful operation on encrypted data without decrypting the data. It was first posed by Rivest, Adleman and Dertouzos in [Rivest et al., 1978]:

> Consider a small loan company which uses a commercial time-sharing service to store its records. The loan company's "data bank" obviously contains sensitive information which should be kept private. On the other hand, suppose that the information protection techniques employed by the time sharing service are not considered adequate by the loan company. In particular, the systems programmers would presumably have access to the sensitive information. The loan company therefore decides to encrypt all of its data kept in the data bank and to maintain a policy of only decrypting data at the home office – data will never be decrypted by the time-shared computer.

An encryption scheme is homomorphic if for any two plaintext x and y:

> $Enc_{pk}(x) \bigotimes Enc_{pk}(y) = Enc_{pk}(x \bigoplus y)$ where Enc is encryption function, pk is the public key, $\bigotimes$ is operation on ciphertext, and $\bigoplus$ is operation on plaintext.

These two operators $\bigotimes$ and $\bigoplus$ are very specific. If a cryptosystem that supports an arbitrary function f on ciphertext, then it is called fully homomorphic cryptosystem:

$$f(Enc_{pk}(m_1), Enc_{pk}(m_2), ..., Enc_{pk}(m_k) = Enc_{pk}(f(m_1, m_2, ..., m_k))$$

The first fully homomorphic encryption system was proposed by Craig Gentry [Gentry, 2009]; however, in this thesis we are mostly concern with partially homomorphic encryption (either additive or multiplicative, but not both), specifically additive ElGamal, so we are not going to present the details overview of Craig Gentry fully homomorphic construction. From now on, we would be using the term homomorphic encryption for partially homomorphic encryption.

Now, keeping in mind that homomorphic encryption enables us to perform useful operation on encrypted data, we will see what kind of homomorphic property is exhibited by the ElGamal method discussed in the previous section. Given a public infrastructure (G, p, g, h) for ElGamal scheme, we encrypt two message m_1 and m_2 by taking two random numbers r_1, r_2 from the group:

$$Enc(m_1, r_1) := (g^{r_1}, m_1 * h^{r_1})$$

$$Enc(m_2, r_2) := (g^{r_2}, m_2 * h^{r_2})$$

If we multiply these two ciphers together pairwise, we get $(g^{r_1+r_2}, m_1 * m_2 * h^{r_1+r_2})$. After decrypting this combined ciphertext, we will get $m_1 * m_2$. In this scheme, $\bigotimes$ is multiplication $*$ and $\bigoplus$ is also multiplication $*$. Furthermore, if our end goal is to achieve multiplication on a bunch of plaintext, rather than decrypting the corresponding ciphertext individually and multiplying them, we could simply multiply all the ciphertext together and decrypt the final result. The advantage of this scheme is that it does not leak the individual values which, sometimes, is a very crucial property in many application, specifically in electronic voting. In electronic voting protocols, we do not want to reveal the choices of an individual voter, but it is acceptable to reveal the final tally. However, this scheme is not suitable for electronic voting schemes because it is multiplicative. Almost, to the best of my knowledge, all the electronic voting scheme calculate the finally tally by adding the individual choices of all voters, so the requirement is achieve the addition on plaintext. There are many additive homomorphic encryption schemes, e.g. Benaloh cryptosystem [Benaloh, 1994], Paillier cryptosystem [Paillier, 1999], etc. In addition, we can modify the ElGamal encryption scheme to make additive. In additive case, it works as:

$$Enc(m_1, r_1) := (g^{r_1}, g^{m_1} * h^{r_1})$$

$$Enc(m_2, r_2) := (g^{r_2}, g^{m_2} * h^{r_2})$$

Multiplying these two ciphers pairwise would give us, $(g^{r_1+r_2}, g^{m_1+m_2} * h^{r_1+r_2})$ which would decrypt as $g^{m_1+m_2}$. We can calculate the value of $m_1 + m_2$ by using linear search algorithm, or more efficient one Pohlig–Hellman algorithm [Pohlig and Hellman, 2006]. However, the downside of this scheme is that if the value of $m_1 + m_2 + \cdots + m_n$ (assuming n values) is very large, calculating it from $g^{m_1+m_2+\cdots+m_n}$ is not very practical [Cramer et al., 1997].

3.2.5 Zero-Knowledge Proof

In conventional mathematics, a proof of mathematical statement is collection of basic axioms combined according to rules of the system. For example, we want to prove that for any group G with group operation *, for any two elements $x\ y \in G$, we have:

$$(x * y)^{-1} = y^{-1} * x^{-1}$$

Proof: we assume arbitrary x, y. We show that $(x * y)$ and $y^{-1} * x^{-1}$ are inverse of each other by combining them together using the group operator $*$ and using the group laws lead to the identity of the group G.

$$\begin{aligned}
(x * y) * (y^{-1} * x^{-1}) &= x * y * y^{-1} * x^{-1} (associativity) \\
&= x * (y * y^{-1}) * x^{-1} (associativity) \\
&= x * e * x^{-1} (inverses) \\
&= x * x^{-1} (identity) \\
&= e (inverse)
\end{aligned} \tag{3.1}$$

Similarly, we can prove that $(y^{-1} * x^{-1}) * (x * y) = e$. We can also formalize it inside theorem prover and prove it more formally (below is a proof in Coq theorem prover where $*$, the group operation, is represented as f and $^{-1}$, the inverse operation, is represented as *inv*).

```
Lemma inv_distr : forall a b, inv (f a b) = f (inv b) (inv a).
Proof.
  intros a b. symmetry.
```

```
    apply inv_uniq_l.
    rewrite <- assoc.
    rewrite  (assoc (inv b) (inv a) a).
    rewrite (inv_l a).
    rewrite (assoc (inv b) e b).
    rewrite (id_l b).
    rewrite (inv_l b). auto.
  Qed.
```

If a verifier wants to verify the correctness of our proof, she would simply check that if the group rules are applied correctly. Moreover, these proofs are static in nature, i.e. once the prover has produced the proof, the content of proof is not going to change over time, and there would not be any interaction between prover and verifier if verifier wants to verify the proof. In addition, the verifier not only learned that the statement is true, but she also learned the content of proof (gained some knowledge).

In contrast, zero-knowledge-proof, first introduced by Goldwasser, Micali, and Rackoff [Goldwasser et al., 1985] , is a probabilistic proof system that involves an explicit notion of interaction between a prover and a verifier. In addition, the goal of the prover is to convince the verifier about the validity of some statement without revealing any information, i.e. the only thing the verifier would learn is that if statement is true or false without any other information. More formally, zero-knowledge proof for a language $L \in \{0,1\}^*$ (generally NP) is an interactive proof between a (computationally unbounded) prover P and a (polynomial time) verifier V. Furthermore, the goal of P is to convince V that $x \in L$ such that:

Completeness: If $x \in L$ then the honest prover P would convince the honest verifier V to accept the claim with overwhelming probability. If P can always convince (probability 1) the V that $x \in L$, the proof system has perfect completeness.

Soundness: If $x \notin L$ then dishonest prover P^* can not convince the honest verifier V to accept the claim (with some small probability error known as soundness error)

Zero-Knowledge: A malicious verifier V^* would gain no additional information by interacting with an honest prover P other than $x \in L$. More formally, for every (polynomial time) program V^* there exists a (polynomial time) program S, also known as simulator, which can produce the transcript of protocol by itself without interacting with anyone. Moreover, the transcript produced by simulator S is indistinguishable from real transcript, produced by interaction between the prover and the verifier.

3.2.5.1 Zero-Knowledge Proof of Knowledge

Sometimes, the fact that $x \in L$ is completely trivial [Bellare and Goldreich, 1993]. For example, for any given finite group G of order p (p is prime), a random element h from the group G, and generator g of the group G, a prover claims that there is a x such that $g^x = h$. This is trivial because we know that there always exists such x (because $h \in G$) ; however, the challenge is to show that the prover knows the witness x. Formally, zero-knowledge proof of knowledge is defines as: let $R = (x, w) \subset L \times W$ is a binary relation such that $x \in L$ is common string between prover P and verifier V and $w \in W$, also known as witness, is private to the prover P. Moreover, the goal of prover P is to convince verifier V that $(x, w) \in R$ in zero-knowledge, i.e. without revealing anything else other than showing that the statement $(x, w) \in R$ is true.

3.2.6 Sigma Protocol

Sigma protocols [Cramer et al., 1994] are efficient way to achieve zero-knowledge proof of knowledge. Sigma protocol is a three step communication between a prover P and a verifier V where goal of the prover is to convince the verifier that she knows witness w for common input x such that $(x, w) \in R$:

1. P sends a message a
2. V sends a random string e
3. P replies with z

Based on public inputs (x, a, e, z), the verifier V decides to accept or reject the proof. A protocol is said to be sigma protocol for a relation R if:

Completeness: when prover and verifier follow the protocol for public input x and (private) witness w, verifier accepts the proof

Special Soundness: For a given public input x, if prover can produce two accepting transcript (a, e, z) and (a, e', z') (e and e' are disjoint), there exists an efficient program, extractor, which can extract the witness w.

Honest Verifier Zero-Knowledge: For a given public input x and random input e, there is a simulator which outputs an accepting transcript (a, e, z) which is indistinguishable from a proof generated by a prover interacting with honest verifier.

A concrete example of sigma protocol [Cramer et al., 1994] is Schnorr protocol [Schnorr, 1990]. In this example, the goal of a prover P is to prove the knowledge of discrete logarithm in a group of order q (q is prime) to a verifier V. Furthermore, g is the generator of group G, x is the public input and w is private input with relation $x = g^w$. The protocol follows:

- Prover P randomly selects an element r from Z_q, computes $a = g^r$ and sends a to verifier V
- Verifier V randomly selects an element c from Z_q and sends it to P
- Prover P sends $z = r + c * w$ to V. V checks $g^z = a * x^c$

For the protocol described above, all three properties, completeness, special soundness, and honest verifier zero-knowledge, hold.

- Completeness holds with probability 1. Simplifying the expression g^z shows that it is equal to $a * x^c$. Replacing the z by $r + c * w$ in expression g^z, we get g^{r+c*w}. Using addition rule of power, g^{r+c*w} can be simplified as $g^r * g^{c*w}$. First step of protocol, $a = g^r$, so we can replace the $g^r * g^{c*w}$ by $a * g^{c*w}$. From the group infrastructure, we have $x = g^w$, so we can write x at place of g^w, therefore, $a * g^{c*w}$ transforms into $a * x^c$.
- Special soundness holds. For any two given response, $z_1 = r + w * c_1$ and $z_2 = r + w * c_2$, we can find the witness w by $(z_2 - z_1)/(c_2 - c_1)$.
- Honest verifier zero-knowledge also holds. Simulator can always produce a transcript $(g^z x^{-c}, c, z)$ by randomly choosing c (the random choice c is the reason for special honest verifier zero-knowledge), and z.

Fiat-Shamir Transform: In practice, the *Fiat-Shamir* [Fiat and Shamir, 1987] transform is used to turn a Sigma protocol into a non-interactive proof. As a consequence, there is no longer any interaction with verifier.

- Prover P randomly selects an element r from Z_q, computes $a = g^r$
- Prover P computes $c = H(a||x)$ where H is a hash function and $||$ is concatenation function.
- Prover P computer $z = r + c * w$

Finally, P publishes the transcript (a, c, z) for anyone to verify her claim. Subsequently, any one who is verifying the claim has to check two things: (i) $c := H(a||x)$, and (ii) $g^z = a * x^c$.

3.2.7 Commitment Schemes

Commitment schemes are cryptographic primitives equivalent to real life sealed lock-box. Once the lock-box is locked and sealed, the content inside can not be changed without breaking the lock and seal. In general, commitment primitives are backbone of any cryptographic protocol between two parties, communicating over the Internet, to force them to follow the protocol honestly, even they would have a huge gain from deviating from the protocol. For example, in order to save some time before a match, Indian cricket team captain (the coin tossing captain), living in Delhi, and Australian cricket team captain (the calling captain), living in Canberra, decide to toss a coin in advance over the Internet, using a mobile application called toss-app, for an upcoming series of one-day matches [2]. Assuming the workings of toss-app is naive and all the messages posted in the chat box of toss-app are in plaintext. Using the toss-app, both captains, the coin tossing captain and the calling captain, post their outcome in the chat box, and the toss winner is decided based on the messages posted by the two captains. How likely would be the case where both captains are honest, if the toss plays a major role in winning the series? We can not expect them to be honest because they both have incentive (winning the series) to cheat during the protocol (coin toss).

The question is can we devise some scheme which would force the both parties to behave honestly? The answer is yes, we can devise such scheme. We would use sealed lock-box concept, albeit digital one. Moreover, the first captain would put his call in a digitally sealed lock-box and post it in the chat box. Because it is sealed and locked, the other captain would have no idea what is the content inside it. Furthermore, it is impossible to break the

[2] In a cricket match, which is very popular sport in India and Australia, both captains meet in the ground and toss a coin to decide who would have the first call.

lock-box, so it is fruitless and waste of time for the other captain to even try. The other captain will toss the coin and post the outcome in a separate digital sealed lock-box in the chat box. Now that there are two digital sealed lock-box, which can only be opened by the respective owners, posted in the chat box, the captains would move for the next phase of coin tossing called reveal phase. In the reveal phase, they both would open their sealed locked box to show that what they have locked, and the decision would be taken accordingly [3].

Formally, a commitment scheme is three step protocol between a sender S and a receiver R:

1. Commit phase: sender S commits a message m by generating a random number r and using some algorithm C, which takes the message and random r. Moreover, the committed value produced by the commitment algorithm C, $c = C(m, r)$, is shared with receiver R.
2. Reveal phase: In the reveal phase, the sender reveals the message m and randomness r which are subsequently used by receiver to verify the result, i.e. the receiver computes $c' = C(m, r)$ and matches it again the given c in the commit phase of protocol.

Security Properties: Commitment schemes have two properties: hiding and binding. Hiding property ensures that the receiver can not recover or recompute the original message m from the given commitment c, i.e. it forces the receiver to behave honestly in the protocol. Furthermore, binding property ensures that it is impossible for sender to come up with another message m' which is different from m but produces the same commitment c, i.e. it forces the sender to behave honestly in the protocol.

Pedersen commitment: Finally, we give a brief overview of a Pedersen commitment scheme which is based on discrete logarithm. The protocol as follows assuming the public parameter available to sender and receiver, i.e. the set up has been conducted to generate the the public parameter, and both parties have these values. These values include a prime p, y a randomly chosen element from Z_p^*, and g a randomly chosen generator from Z_p^*.

- Commit phase: The sender generates a random r from Z_p^*, computes commitment $c = g^r * y^m$ and sends the commitment to receiver

[3] Story influenced by Manuel Blum's coin flipping by telephone

- Verification phase: In verification phase, the sender reveals the original message m and the randomness r. Finally, the receiver computes $g^r * y^m$. If the computed value matches with the commitment received in commitment phase, then she accepts it otherwise reject it.

3.3 Summary

In this chapter, we gave a brief summary of Coq theorem prover and cryptographic primitives needed to understand the further chapters. By no means, these descriptions were exhaustive. For a detailed treatment of Coq theorem prover, [Bertot et al., 2004] [Chlipala, 2013] can be referred, and for cryptography, [Menezes et al., 1996] [Schneier, 1995] [Paar and Pelzl, 2009] can be referred. In the next chapter, we will discuss the Schulze method, and the machinery for its formalization.

Chapter 4

Schulze Method : Evidence Carrying Computation

The negligence of a few could easily send a ship to the bottom, but if it has the wholehearted co-operation of all on board it can be safely brought to part.

Sardar Vallabhbhai Patel

4.1 Introduction

Correctness and verifiability/evidence are two main pillars of any democratic election. In case of paper ballot election, correctness and verifiability of counting is achieved by public scrutiny because each step is carefully observed by general member of public, and agents from different political parties. For example, casting ballot at booth is carefully observed by polling agents and counting ballots is observed by the scrutineers (Figure 4.1) appointed by different political parties. Given that electronic voting is relatively young, in this chapter we investigate how to achieve the correctness and verifiability similar to paper ballot election.

Chapter overview: In this chapter, we explain the Schulze method in section 4.2, and its formal specification in section 4.3. The corner stone of our formalisation is a correct-by-construction dependent inductive data type that represents all correct executions (4.3.1) with the formal proof of that ev-

Figure 4.1: Scrutineers, in green jacket, observing the ballot counting

ery Schulze election has winners (4.3.2). Every inhabitant of this dependent inductive data type not only produces a final result but also all the intermediate steps which lead to the notion of evidence or scrutiny sheet (section 4.4). In section 4.5, we discuss the optimization techniques to overcome the deficiencies in extracted Haskell code from Coq formalization. Based on these optimizations, the extracted Haskell code was able to count millions of ballots in few minutes. Finally, we conclude the chapter in section 4.6 with the achievements and drawbacks of our work on the scale of *Correctness*, *Privacy*, and *Verifiability*.

4.2 Schulze Method

The Schulze Method [Schulze, 2011] is a vote counting scheme that elects a single winner, based on preferential votes. The method itself rests on the relative *margins* between two candidates, i.e. the number of voters that prefer one candidate over another. The margin induces an ordering between candidates, where a candidate c is more preferred than d, if more voters prefer c over d than vice versa. One can construct simple examples (see e.g. [Rivest and Shen, 2010]) where this order does not have a maximal element (a so-called *Condorcet Winner*). Schulze's observation is that this ordering *can* be made transitive by considering sequences of candidates (called *paths*). Given candidates c and d, a *path* between c and d is a sequence of candidates $p = (c, c_1, \ldots, c_n, d)$ that joins c and d, and the *strength* of a path is the minimal margin between adjacent nodes. This induces the *generalised margin* between candidates c and d as the strength of the strongest path that joins c and d. A candidate c then wins a Schulze count if the generalised margin between c and any other candidate d is at least as large as the generalised margin between d and c. More concretely:

- Consider an election with a set of t candidates $C = \{c1, \ldots, ct\}$, and a set of n votes $P = \{b1, \ldots, bn\}$. A vote is represented as function $b : C \to \mathbb{N}$ that assigns natural number (the preference) to each candidate. We recover a strict linear preorder $<_b$ on candidates by setting $c <_b d$ if $b(c) > b(d)$, i.e. c is less preferred over d if the natural number $b(c)$ is greater than the natural number $b(d)$.

- Given a set of ballots P and candidate set C, we construct graph G based on the margin function $m : C \times C \to \mathbb{Z}$. Given two candidates $c, d \in C$, the *margin* of c over d is the number of voters that prefer c over d, minus the number of voters that prefer d over c. In symbols:

$$m(c, d) = \sharp\{b \in P \mid c >_b d\} - \sharp\{b \in P \mid d >_b c\}$$

  where $\sharp$ denotes cardinality and $>_b$ is the strict (preference) ordering given by the ballot $b \in P$.

- A directed *path* in the graph, G, from candidate c to candidate d is a sequence $p \equiv c_0, \ldots, c_{w+1}$ of candidates with $c_0 = c$ and $c_{w+1} = d$ ($w \geq 0$), and the *strength*, st, of path, p, is the minimum margin of adjacent nodes, i.e.

$$st(c_0, \ldots, c_{w+1}) = \min\{m(c_i, c_{i+1}) \mid 0 \leq i \leq w\}.$$

- For candidates c and d, let $M(c, d)$ denote the maximum strength, or generalised margin of a path from c to d i.e.

$$M(c, d) = \max\{st(p) : p \text{ is path from } c \text{ to } d \text{ in } G\}$$

- The winning set, formally defined in 4.3.2, is defined as

$$W = \{c \in C : \forall d \in C \setminus \{c\}, M(c,d) \geq M(d,c)\}$$

In other words, the Schulze method stipulates that a candidate $c \in C$ is a *winner* of the election with margin function m if, for all other candidates $d \in C$, there exists a number $k \in \mathbb{Z}$ such that

- there is a path p from c to d with strength $st(p) \geq k$
- all paths q from d to c have strength $st(q) \leq k$.

Informally speaking, we can say that candidate c *beats* candidate d if there is a path p from c to d which is stronger than any path from d to c. Using this terminology, a candidate c is a winner if c cannot be beaten by any (other) candidate.

4.2.1 An Example

Suppose that for some given set of ballots (the actual set of ballots are not very important because we want to demonstrate the Condercet Paradox that we will explain below) for a given set of candidates $\{A, B, C\}$, we have computed the margin function m such that m (A, B) = 3, m (B, A) = -3, m (A, C) = -1, m (C, A) = 1, m (B, C) = 5, and m (C, B) = -5. We have drawn the graph below (Figure 4.2), and it shows that collective preferences can be cyclic, even if the preferences of individual voters are not cyclic. This phenomena is known as Condercet paradox and first observed by french philosopher Marquis de Condorcet in late 18th century [1].

The main idea of the method is to resolve cycles by considering *transitive preferences* or a generalised notion of margin. Figure 4.3 shows the graph interpretation of the generalised margin, M, after running the Schulze method on the margin function m (the word margin function is used interchangeably with margin matrix). In order to compute $M(A, B)$, we first compute all the paths from candidate A to B. Here we have just two paths from A to B, a direct path between them and an intermediate path via candidate C. Now that we have all the paths, we compute the path strength st for each path,

[1] https://gallica.bnf.fr/ark:/12148/bpt6k417181

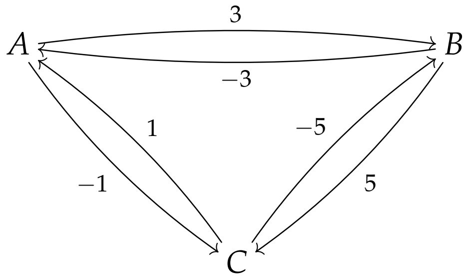

Figure 4.2: Margin Function/Matrix (Graph Interpretation)

$st(A,B) = min\{m(A,B)\}$ and $st(A,C,B) = min\{m(A,C), m(C,B)\}$. Simply these expressions:

$$\begin{aligned} st(A,B) &= \min\{m(A,B)\} \\ &= \min\{3\} \\ &= 3 \end{aligned}$$

$$\begin{aligned} st(A,C,B) &= \min\{m(A,C), m(C,B)\} \\ &= \min\{-1, -5\} \\ &= -5 \end{aligned}$$

Once we have the path strength for every path between A and B, we compute generalised margin $M(A,B) = \max\{st(A,B), st(A,C,B)\}$.

$$\begin{aligned} M(A,B) &= \max\{st(A,B), st(A,C,B)\} \\ &= \max\{3, -5\} \\ &= 3 \end{aligned}$$

Since $M(A,B) = 3$, hence the arrow going A to B has strength 3. Similarly, we can compute other values as well.

Strength of paths between B to A:

$$\begin{aligned} st(B,A) &= \min\{m(B,A)\} \\ &= \min\{-3\} \\ &= -3 \end{aligned}$$

$$\begin{aligned} st(B,C,A) &= \min\{m(B,C), m(C,A)\} \\ &= \min\{5,1\} \\ &= 1 \end{aligned}$$

Generalised margin between B and A:

$$\begin{aligned} M(B,A) &= \max\{st(B,A), st(B,C,A)\} \\ &= \max\{-3,1\} \\ &= 1 \end{aligned}$$

Strength of paths between A to C:

$$\begin{aligned} st(A,C) &= \min\{m(A,C)\} \\ &= \min\{-1\} \\ &= -1 \end{aligned}$$

$$\begin{aligned} st(A,B,C) &= \min\{m(A,B), m(B,C)\} \\ &= \min\{3,5\} \\ &= 3 \end{aligned}$$

Generalised margin between A and C:

$$\begin{aligned} M(A,C) &= \max\{st(A,C), st(A,B,C)\} \\ &= \max\{-1,3\} \\ &= 3 \end{aligned}$$

Strength of paths between C to A:

$$\begin{aligned} st(C,A) &= \min\{m(C,A)\} \\ &= \min\{1\} \\ &= 1 \end{aligned}$$

$$\begin{aligned} st(C,B,A) &= \min\{m(C,B), m(B,A)\} \\ &= \min\{-5,-3\} \\ &= -5 \end{aligned}$$

Generalised margin between C and A:

$$\begin{aligned} M(C,A) &= \max\{st(C,A), st(C,B,A)\} \\ &= \max\{1,-5\} \\ &= 1 \end{aligned}$$

Strength of paths between C to B:

$$\begin{aligned} st(C,B) &= \min\{m(C,B)\} \\ &= \min\{-5\} \\ &= -5 \end{aligned}$$

$$\begin{aligned} st(C,A,B) &= \min\{m(C,A), m(A,B)\} \\ &= \min\{1,3\} \\ &= 1 \end{aligned}$$

Generalised margin between C and B:

$$\begin{aligned} M(C,B) &= \max\{st(C,B), st(C,A,B)\} \\ &= \max\{-5,1\} \\ &= 1 \end{aligned}$$

Strength of paths between B to C:

$$\begin{aligned} st(B,C) &= \min\{m(B,C)\} \\ &= \min\{5\} \\ &= 5 \end{aligned}$$

$$\begin{aligned} st(B,A,C) &= \min\{m(B,A), m(A,C)\} \\ &= \min\{-3,-1\} \\ &= -3 \end{aligned}$$

Generalised margin between B and C:

$$\begin{aligned} M(B,C) &= \max\{st(B,C), st(B,A,C)\} \\ &= \max\{5,-3\} \\ &= 5 \end{aligned}$$

Now we have computed all the values of generalised margin M, we can interpret it as a graph show below. It is clear from the graph that candidate A is winner, as she beats B with strength 3 (reverse path from B to A is weaker, i.e. strength 1) and C with strength 3 (reverse path from C to A is weaker, i.e. strength 1).

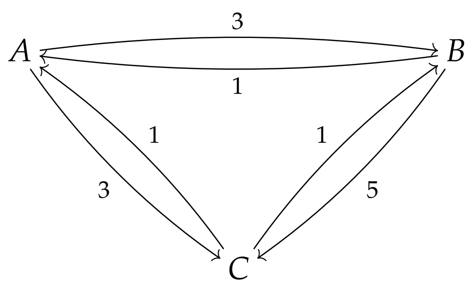

Figure 4.3: Generalised Margin (Graph Interpretation)

4.3 Formal Specification

We start our Coq formalization assuming finite and non-empty set of candidates. Also, we assume decidable equality on candidates. For our purposes, the easiest way of stipulating that a type be finite is to require existence of a list containing all inhabitants of this type [Firsov and Uustalu, 2015b].

```
Variable cand : Type.
Variable cand_all : list cand.
Hypothesis cand_fin : forall c: cand, In c cand_all.
Hypothesis dec_cand : forall n m : cand, {n = m} + {n <> m}.
Hypothesis cand_in : cand_all <> [].
```

For the specification of winners of Schulze elections, we take the margin function as given for the moment (and later construct it from the incoming ballots). In Coq, this is conveniently expressed as a variable:

```
Variable marg : cand -> cand -> Z.
```

We formalise the notion of path and strength of a path by means of a single (but ternary) inductive proposition that asserts the existence of a path of strength $\geq k$ between two candidates, for $k \in \mathbb{Z}$. The notion of winning candidate is that it beats every other candidate, i.e. all the paths from the winner to other candidates are at least as strong as the reverse path. Dually, the notion of loser is that there is a candidate who beats the loser, i.e. the path from the candidate to the loser is stronger than the reverse path.

```
(* prop-level path *)
Inductive Path (k: Z) : cand -> cand -> Prop :=
  | unit c d : marg c d >= k -> Path k c d
  | cons  c d e : marg c d >= k ->
         Path k d e -> Path k c e.

(* winning condition of Schulze Voting *)
Definition wins_prop (c: cand) :=
  forall d: cand, exists k: Z,
    Path k c d /\ (forall l, Path l d c -> l <= k).
```

```
(* dually, the notion of not winning: *)
Definition loses_prop (c : cand) :=
  exists k: Z, exists  d: cand,
    Path k d c /\ (forall l, Path l c d -> l < k).
```

We reflect the fact that the above are *propositions* in the name of the definitions, in anticipation of type-level definitions of these notions later. The reason for having a *Prop* level definition is that it is very easy and intuitive for human to inspect the definitions, and ascertain the correctness of formalization. As we discussed in the **Type vs. Prop** (section 3.1.3), the main reason for having an equivalent type-level versions of the above is that purely propositional information is discarded during program extraction, unlike the type-level notions of winning and losing that represent evidence of the correctness of the determination of winners. Our goal is to not only compute winners and losers according to the definition above, but also to provide independently verifiable evidence, a scrutiny sheet or certificate, of the correctness of our computation. The propositional definitions of winning and losing above serve as a reference to calibrate their type level counterparts, and we demonstrate the equivalence between propositional and type-level conditions in the next section.

One of the fundamental question about declaring someone as a winner or loser is that how can we know that, say, a candidate c in fact wins a Schulze election, and that, say, d is not a winner? One possible answer is simply re-run an independent implementation of the method (usually hoping that results would be confirmed). But what happens if results diverge?

One major aspect of our work is that we can answer this question by not only computing the set of winners, but in fact presenting *evidence* for the fact that a particular candidate does or does not win. This is a re-emphasis on *Correctness*, and convincing to all, specifically to losers, leaving no ground for speculation. As we stated earlier that in the context of electronic vote counting, this is known as a *scrutiny sheet, or certificate*: a tabulation of all relevant data that allows us to verify the election outcome. Again drawing on an already computed margin function, to demonstrate that a candidate c wins, we need to exhibit an integer k for all competitors d, together with

- evidence for the existence of a path from c to d with strength $\geq k$
- evidence for the non-existence of a path from d to c that is stronger than k

The first item is straight forward, as a path itself is evidence for the existence of a path, and the notion of path is inductively defined. For the second item, we need to produce evidence of membership in the *complement* of an inductively defined set.

Mathematically, given $k \in \mathbb{Z}$ and a margin function $m : C \times C \to \mathbb{Z}$, the pairs $(c, d) \in C \times C$ for which there exists a path of strength $\geq k$ that joins both are precisely the elements of the least fixpoint $LFP(V_k)$ of the monotone operator $V_k : Pow(C \times C) \to Pow(C \times C)$ (Pow stands for powerset), defined by

$$V_k(R) = \{(c, e) \in C \times C \mid m(c, e) \geq k \text{ or } (m(c, d) \geq k \text{ and } (d, e) \in R \text{ for some } d \in C)\}$$

where R is a subset of $C \times C$, i.e. $R \subseteq C \times C$. It is easy to see that this operator is indeed monotone, and that the least fixpoint exists, e.g. using Kleene's theorem [Stoltenberg-Hansen et al., 1994]. To show that there is *no* path between d and c of strength $> k$, we therefore need to establish that $(d, c) \notin LFP(V_{k+1})$.

By duality between least and greatest fixpoints, we have that

$$(c, d) \in C \times C \setminus LFP(V_{k+1}) \iff (c, d) \in GFP(W_{k+1})$$

where for arbitrary k, $W_k : Pow(C \times C) \to Pow(C \times C)$ is the operator dual to V_k, i.e.

$$W_k(R) = C \times C \setminus (V_k(C \times C \setminus R))$$

and $GFP(W_k)$ is the greatest fixpoint of W_k. As a consequence, to demonstrate that there is *no* path of strength $> k$ between candidates d and c, we need to demonstrate that $(d, c) \in GFP(W_{k+1})$. By the Knaster-Tarski fixpoint theorem [Tarski, 1955], this greatest fixpoint is the supremum of all W_{k+1}-coclosed sets, that is, sets $R \subseteq C \times C$ for which $R \subseteq W_{k+1}(R)$. That is, to demonstrate that $(d, c) \in GFP(W_{k+1})$, we need to exhibit a W_{k+1}-coclosed set R with $(d, c) \in R$. If we unfold the definitions, we have:

$$W_k(R) = \{(c, e) \in C \times C \mid m(c, e) < k \text{ and } (m(c, d) < k \text{ or } (d, e) \in R \text{ for all } d \in C)\}$$

so that given *any* fixpoint R of W_k and $(c, e) \in W$, we know that (i) the margin between c and e is $< k$ so that there's no path of length 1 between c and e, and (ii) for any choice of midpoint d, either the margin between c and d is $< k$ (so that $c, d, \ldots$ cannot be the start of a path of strength $\geq k$) or we don't have a path between d and e of strength $\geq k$. We use the following terminology:

Definition 1 *Let $R \subseteq C \times C$ be a subset and $k \in \mathbb{Z}$. Then R is W_k-*coclosed*, or simply k-*coclosed*, if $R \subseteq W_k(R)$.*

Mathematically, the operator W_k acts on subsets of $C \times C$ that we think of as predicates. In Coq, we formalise these predicates as boolean valued functions and obtain the following definitions where we isolate the function `marg_lt` (that determines whether the margin between two candidates is less than a given integer) for clarity:

```
Definition marg_lt (k : Z) (p : (cand * cand)) :=
  Zlt_bool (marg (fst p) (snd p)) k.

Definition W (k : Z) (p: cand * cand -> bool)
  (x: cand * cand) := andb  (marg_lt k x)
  (forallb (fun m => orb (marg_lt k (fst x, m))
                         (p (m, snd x))) cand_all).
```

In order to formulate type-level definitions, we need to promote the notion of path from a Coq proposition to a proper type, and formulate the notion of k-coclosed predicate.

```
Definition coclosed (k : Z) (f : (cand * cand) -> bool) :=
  forall x, f x = true -> W k f x = true.

Inductive PathT (k: Z) : cand -> cand -> Type :=
| unitT c d :  marg c d >= k -> PathT k c d
| consT c d e : marg c d >= k ->
        PathT k d e -> PathT k c e.
```

Now, we have following type-level definition of winning (and dually, no-winning) for Schulze counting. As we see that these definitions not only produces the result, but they also produce witness, e.g. the `wins_type` definition states that if a candidate, say c, is the winner, for each individual candidate participating in election, it produces two witnesses: (i) a path from itself to the beating candidate of certain strength, say k, and (ii) a k+1 coclosed set. These witnesses are basic building blocks of the scrutiny sheet we produce after election.

```
Definition wins_type c :=
  forall d : cand, existsT (k : Z),
  ((PathT k c d) * (existsT (f : (cand * cand) -> bool),
   f (d, c) = true /\ coclosed (k + 1) f))%type.

Definition loses_type (c : cand) :=
  existsT (k : Z) (d : cand),
  ((PathT k d c) * (existsT (f : (cand * cand) -> bool),
    f (c, d) = true /\ coclosed k f))%type.
```

We have two definitions of winning, `wins_prop` which is easier for a human to inspect; on the other hand, `wins_type` which is useful for the machine. We close the gap by formally establishing that type level winning and prop level winning (dually, not winning) are in fact equivalent.

```
Lemma wins_type_prop :
   forall c, wins_type c -> wins_prop c.

Lemma wins_prop_type :
  forall c, wins_prop c -> wins_type c.

Lemma loses_type_prop :
  forall c, loses_type c -> loses_prop c.

Lemma loses_prop_type :
  forall c, loses_prop c -> loses_type c.
```

The different nature of the two propositions does not allow us to claim an equivalence between both notions, as Coq defines bi-implication only on propositions.

The proof of the first statement, *wins_type_prop*, is completely straight forward, as the type, *win_type*, carries all the information needed to establish the propositional winning, *wins_prop*. However, for the second statement *wins_prop_type*, Coq does not allow the case analysis or induct on a term of sort `Prop` when the sort of goal is not in `Prop`. We follow the techniques that we have described in `Type vs Prop` section (3.1.3.1). To prove the second statement, we first introduced an intermediate lemma based on the *iterated margin function* $M_k : C \times C \to Z$. Intuitively, $M_k(c, d)$ is the strength of the strongest path between c and d of length $\leq k+1$. Formally, $M_0(c,d) = m(c,d)$ and

$$M_{i+1}(c,d) = \max\{M_i(c,d), \max\{\min\{m(c,e), M_i(e,d) \mid e \in C\}\}$$

for $i \geq 0$. It is intuitively clear (and we establish this fact formally) that the iterated margin function stabilises at the n-th iteration (where n is the number of candidates), as paths with repeated nodes don't contribute to maximising the strength of a path. This proof loosely follows the evident pen-and-paper proof given for example in [Carré, 1971] that is based on cutting out segments of paths between repeated nodes and so reaches a fixed point.

```
Lemma iterated_marg_fp: forall (c d : cand) (n : nat),
     M n c d <= M (length cand_all) c d.
```

That is, the *generalised margin*, i.e. the strength of the strongest (possibly infinite) path between two candidates is effectively computable.

This allows us to relate the propositional winning conditions to the iterated margin function and showing that a candidate c is winning implies that the generalised margin between this candidate and any other candidate d is at least as large as the generalised margin between d and c.

```
Lemma wins_prop_iterated_marg (c : cand) : wins_prop c ->
   forall d, M (length cand_all) d c <=
   M (length cand_all) c d.
```

This condition on iterated margins can in turn be used to establish the type-level winning condition, thus closing the loop to the type level winning condition.

```
Lemma iterated_marg_wins_type (c : cand) : (forall d,
    M (length cand_all) d c <= M (length cand_all) c d)
    -> wins_type c.
```

Similarly, we connect the propositional losing to type level losing via generalised margin. We show that candidate c is losing then there is a candidate d and generalised margin between candidate d and c is more that generalised margin between c and d. Using this fact, we can prove the type level losing condition.

```
Lemma loses_prop_iterated_marg (c : cand):
    loses_prop c ->
    (exists d, M (length cand_all) c d <
    M (length cand_all) d c).

Lemma iterated_marg_loses_type (c : cand) :
    (exists d, M (length cand_all) c d <
    M (length cand_all) d c)
    -> loses_type c.
```

The proof of lemma *iterated_marg_loses_type* is not straight forward because we are in a similar situation as we were in *wins_prop_type*. We can not eliminate `exists n, P n` in order to show `existsT n, P n`, because Coq would not allow to do case analysis on `exists n, P n` (a term of type `Prop`) since the goal, `existsT n, P n` (a term of type `Type`), is not in `Prop`. We again follow the technique described in `Type vs Prop` section (3.1.3.1). We do a linear search on list of candidates to find the witness constructively, and since, the list of candidates is finite we would eventually terminate and find one. This completes our loop of prop level loser to type level loser.

```
Corollary reify_opponent (c: cand):
  exists  d,
  M (length cand_all) c d < M (length cand_all) d c ->
  existsT d,
  M (length cand_all) c d < M (length cand_all) d c.
```

The crucial part of establishing the type-level winning conditions in the proof of the lemma above is the construction of a coclosed set. First note that `M (length cand_all)` is precisely the generalised margin function. Writing g for this function, we assume that $g(c,d) \geq g(d,c)$ for all candidates d, and given d, we need to construct a $k+1$-coclosed set S where $k = g(c,d)$. One option is to put $S = \{(x,y) \mid g(x,y) < k+1\}$. As every i-coclosed set is also j-coclosed for $i \leq j$, the set $S' = \{(x,y) \mid g(x,y) < g(d,c)+1\}$ is also $k+1$-coclosed and (in general) of smaller cardinality. We therefore witness the existence of a $k+1$-coclosed set with S' as this leads to certificates that are smaller in size and therefore easier to check.

We note that the difference between the type-level and the propositional definition of winning is in fact more than a mere reformulation. As remarked before (3.1.3), one difference is that purely propositional evidence is erased during program extraction so that using just the propositional definitions, we would obtain a determination of election winners, but no additional information that substantiates this (and that can be verified independently). The second difference is conceptual: it is easy to verify that a set is indeed coclosed as this just involves a finite (and small) amount of data, whereas the fact that *all* paths between two candidates don't exceed a certain strength is impossible to ascertain, given that there are infinitely many paths.

In summary, determining that a particular candidate wins an election based on the `wins_type` notion of winning, the extracted program will *additionally* deliver, for all other candidates,

- an integer k and a path of strength $\geq k$ from the winning candidate to the other candidate
- a coclosed set that witnesses that no path of strength $> k$ exists in the opposite direction.

It is precisely this additional data, which we call scrutiny sheet, (on top of merely declaring a set of election winners) that allows for scrutiny of the process, as it provides an orthogonal approach to verifying the correctness of the computation: both checking that the given path has a certain strength, and that a set is indeed coclosed, is easy to verify. We reflect more on this in Section 4.6, and present an example of a full scrutiny sheet in the next section, when we join the type-level winning condition with the construction of the margin function from the given ballots.

4.3.1 Vote Counting as Inductive Type

Up to now, we have described the specification of Schulze voting relative to a given margin function. We now describe the specification (and computation) of the margin function given a profile (set) of ballots. Our formalisation describes an individual *count* as a type with the interpretation that all inhabitants of this type are correct executions of the vote counting algorithm. In the original paper describing the Schulze method [Schulze, 2011], a ballot is a linear preorder over the set of candidates.

In practice, ballots are implemented by asking voters to put numerical preferences against the names of candidates as represented by the Figure 4.4 . The most natural representation of a ballot is therefore a function $b : C \to \mathbb{N}$

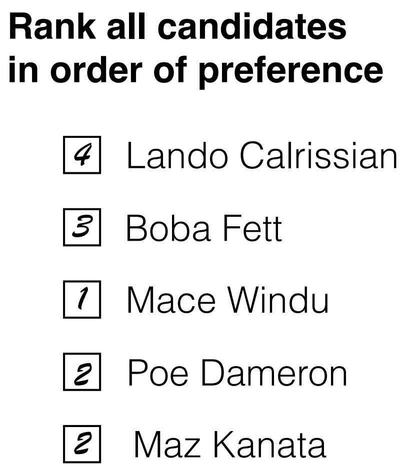

Figure 4.4: Ballot Representation

that assigns a natural number (the preference) for each candidate, and we recover a strict linear preorder $<_b$ on candidates by setting $c <_b d$ if $b(c) > b(d)$.

As preferences are usually numbered beginning with 1, we interpret a preference of 0 as the voter failing to designate a preference for a candidate as this allows us to also accommodate incomplete ballots. This is clearly a design decision, and we could have formalised ballots as functions $b : C \to 1 + \mathbb{N}$ (with 1 being the unit type) but it would add little to our analysis.

```
Definition ballot := cand -> nat.
```

The count of an individual election is then parameterised by the list of ballots cast, and is represented as a dependent inductive type. More precisely, we have a type `State` that represents either an intermediate stage of constructing the margin function or the determination of the final election result:

```
Inductive State: Type :=
| partial: (list ballot * list ballot)  ->
  (cand -> cand -> Z) -> State
| winners: (cand -> bool) ->  State.
```

The interpretation of this type is that a state either consists of two lists of ballots and a margin function, representing

- the set of ballots counted so far, and the set of invalid ballots seen so far
- the margin function constructed so far

or, to signify that winners have been determined, a boolean function that determines the set of winners.

The type that formalises correct counting of votes according to the Schulze method is parameterised by the profile of ballots cast (that we formalise as a list), and depends on the type `State`. That is to say, an inhabitant of the type `Count st`, for `st` of type `State`, represents a correct execution of the voting protocol up to reaching state `st`. This state generally represents intermediate stages of the construction of the margin function, with the exception of the final step where the election winners are being determined. The inductive type takes the following shape:

```
Inductive Count (bs : list ballot) : State -> Type :=
| ax us m : us = bs -> (forall c d, m c d = 0) ->
   (* zero margin *)
  Count bs (partial (us, []) m)
| cvalid u us m nm inbs :
  Count bs (partial (u :: us, inbs) m) ->
   (* u is valid  *)
  (forall c, (u c > 0)%nat) ->
  (forall c d : cand,
    (* c preferred to d *)
    ((u c < u d) -> nm c d = m c d + 1)  /\
    (* c, d rank equal  *)
    ((u c = u d) -> nm c d = m c d)     /\
    (* d preferred to c *)
    ((u c > u d) -> nm c d = m c d - 1))  ->
    Count bs (partial (us, inbs) nm)
| cinvalid u us m inbs :
  Count bs (partial (u :: us, inbs) m) ->
   (* u is invalid     *)
  (exists c, (u c = 0)%nat)  ->
  Count bs (partial (us, u :: inbs) m)
```

```
| fin m inbs w
  (d : (forall c, (wins_type m c) + (loses_type m c))) :
  (*no ballots left*)
  Count bs (partial ([], inbs) m)   ->
  (forall c, w c = true <-> (exists x, d c = inl x)) ->
  (forall c, w c = false <-> (exists x, d c = inr x)) ->
  Count bs (winners w).
```

The intuition here is simple: the first constructor, `ax`, initiates the construction of the margin function, and we ensure that all ballots are uncounted, no ballots are invalid (yet), and the margin function is constantly zero. The second constructor, `cvalid`, updates the margin function according to a valid ballot (all candidates have preferences marked against their name), and removes the ballot from the list of uncounted ballots. The constructor `cinvalid` moves an invalid ballot to the list of invalid ballots, and the last constructor `fin` applies only if the margin function is completely constructed (no more uncounted ballots). In its arguments, `w : cand -> bool` is the function that determines election winners, and `d` is a function that delivers, for every candidate, type-level evidence of winning or losing, consistent with `w`. Given this, we can conclude the count, and declare `w` to be the set of winners (or more precisely, those candidates for which `w` evaluates to `true`).

Together with the equivalence of the propositional notions of winning or losing a Schulze count with their type-level counterparts, every inhabitant of the type `Count b (winners w)` then represents a correct count of ballots `b` leading to the boolean predicate `w : cand -> bool` that determines the winners of the election with initial set `b` of ballots.

The crucial aspect of our formalisation of executions of Schulze counting is that the transcript of the count is represented by a type that is *not* a proposition. As a consequence, extraction delivers a program that produces the (set of) election winner(s), *together* with the evidence recorded in the type to enable independent verification.

4.3.2 All Schulze Elections Have Winners

The main theorem, the proof of which we describe in this section, is that all elections according to the Schulze method engender a boolean-valued function `w : cand -> bool` that determines precisely which candidates are winners of the election, together with type-level evidence of this. Note that a Schulze election can have more than one winner, the simplest (but not the only) example being when no ballots at all have been cast. The theorem that we establish (and later extract as a program) simply states that for every incoming set of ballots, there is a boolean function that determines the election winners, together with an inhabitant of the type `Count` that witnesses the correctness of the execution of the count.

```
Theorem schulze_winners: forall (bs : list ballot),
  existsT (w: cand -> bool) (p : Count bs (winners w)), True.
```

The first step in the proof is elementary: we show that for any given list of ballots we can reach a state of the count where there are no more uncounted ballots, i.e. the margin function has been fully constructed.

```
Lemma all_ballots_counted: forall (bs : list ballot),
   existsT i m, (Count bs (partial ([], i) m)).
```

The second step relies on the iterated margin function already discussed in Section 4.3. As $M_n(c,d)$ (for n being the number of candidates) is the strength of the strongest path between c and d, we construct a boolean function w such that $w(c) = \texttt{true}$ if and only if $M_n(c,d) \geq M_n(d,c)$ for all $d \in C$. We then construct the type-level evidence required in the constructor `fin` using the function (or proposition) `iterated_marg_wins_type` described earlier.

4.4 Scrutiny Sheet and Experimental Results

The crucial aspect of our formalisation is that the vote counting protocol itself is represented as a dependent inductive type that represents all (correct) partial executions of the protocol. A complete execution can then be understood as a state of vote counting where election winners have been determined. Our main theorem, `schulze_winners`, then asserts that an inhabitant of this type exists, for all possible sets of incoming ballots. Crucially, every such inhabitant contains enough information to (independently) verify the correctness of the election result, and can be thought of as a *certificate* for the count. From a computational perspective, we view tallying not merely as a function that delivers a result, but instead as a function that delivers a result, *together* with evidence that allows us to verify correctness. In other words, we augment verified correctness of an algorithm with the means to verify each particular *execution*.

From the perspective of electronic voting, this means that we no longer need to trust the hardware and software (assuming the cast-as-intended and collected-as-cast verifiability) that were employed to obtain the election result, as the generated certificate can be verified independently. In the literature on electronic voting, this is known as (tallied-as-cast) *verifiability* and has been recognised as one of the cornerstones for building trust in election outcomes by electronic voting research community [Chaum, 2004] [Küsters et al., 2011], [Benaloh and Tuinstra, 1994], [Delaune et al., 2010a], [Bernhard et al., 2017].

Coq's extraction mechanism then allows us to turn our main theorem, `schulze_winners` 4.3.2, into a provably correct program. When extracting, all purely propositional information is erased and given a set of incoming ballots, the ensuing program produces an inhabitant of the (extracted) type `Count` that records the construction of the margin function, together with (type level) evidence of correctness of the determination of winners. That is, we see the individual steps of the construction of the margin function (one step per ballot) and once all ballots are exhausted, the determination of winners, together with paths and coclosed sets. The following is the transcript of a Schulze election where we have added wrappers to pretty-print the information content. This is the (full) scrutiny sheet promised in Section 4.3 and concretely it looks follows:

```
V: [A3 B1 C2 D4,..], I: [],
M: [AB:0 AC:0 AD:0 BC:0 BD:0 CD:0]
-------------------------------------------------------
V: [A1 B0 C4 D3,..], I: [],
M: [AB:-1 AC:-1 AD:1 BC:1 BD:1 CD:1]
-------------------------------------------------------
V: [A3 B1 C2 D4,..], I: [A1 B0 C4 D3],
M: [AB:-1 AC:-1 AD:1 BC:1 BD:1 CD:1]
-------------------------------------------------------
                     . . .
-------------------------------------------------------
V: [A1 B3 C2 D4], I: [A1 B0 C4 D3],
M: [AB:2 AC:2 AD:8 BC:5 BD:8 CD:8]
-------------------------------------------------------
V: [], I: [A1 B0 C4 D3],
```

```
M: [AB:3 AC:3 AD:9 BC:4 BD:9 CD:9]
-----------------------------------------------------
winning: A
  for B: path A --> B of strength 3, 4-coclosed set:
    [(B,A),(C,A),(C,B),(D,A),(D,B),(D,C)]
  for C: path A --> C of strength 3, 4-coclosed set:
    [(B,A),(C,A),(C,B),(D,A),(D,B),(D,C)]
  for D: path A --> D of strength 9, 10-coclosed set:
    [(D,A),(D,B),(D,C)]
losing: B
  exists A: path A --> B of strength 3, 3-coclosed set:
    [(A,A),(B,A),(B,B),(C,A),(C,B),(C,C),
     (D,A),(D,B),(D,C),(D,D)]
losing: C
  exists A: path A --> C of strength 3, 3-coclosed set:
    [(A,A),(B,A),(B,B),(C,A),(C,B),(C,C),
     (D,A),(D,B),(D,C),(D,D)]
losing: D
  exists A: path A --> D of strength 9, 9-coclosed set:
    [(A,A),(A,B),(A,C),(B,A),(B,B),(B,C),
     (C,A),(C,B),(C,C),(D,A),(D,B),(D,C),(D,D)]
```

Here, we assume four candidates, `A`, `B`, `C` and `D` and a ballot of the form `A3 B2 C4 D1` signifies that `D` is the most preferred candidate (the first preference), followed by `B` (second preference), `A` and `C`. In every line, we only display the first uncounted ballot (condensing the remainder of the ballots to an ellipsis), followed by votes that we have deemed to be invalid. We display the partially constructed margin function on the right. Note that the margin function satisfies $m(x, y) = -m(y, x)$ and $m(x, x) = 0$ so that the margins displayed allow us to reconstruct the entire margin function. In the construction of the margin function, we begin with the constant zero function, and going from one line to the next, the new margin function arises by updating according to the first ballot. This corresponds to the constructor `cvalid` and `cinvalid` being applied recursively: we see an invalid ballot being set aside in the step from the second to the third line, all other ballots are valid. Once the margin function is fully constructed (there are no more uncounted ballots), we display the evidence provided in the constructor `fin`: we present evidence of winning (losing) for all winning (losing) candidates. In order to actually verify the computed result, a third party observer would have to

1. Check the correctness of the individual steps of computing the margin

function

2. For winners, verify that the claimed paths exist with the claimed strength, and check that the claimed sets are indeed coclosed.

Contrary to re-running a different implementation on the same ballots, our scrutiny sheet provides an *orthogonal* perspective on the data and how it was used to determine the election result.

We have evaluated our approach by extracting the entire Coq development into Haskell, with all types defined by Coq extracted as is, i.e. in particular using Coq's unary representation of natural numbers, and Haskell native integer representation. The results are displayed in Figure 4.5, and Figure 4.6 using a logarithmic scale. As the reader can see that the execution time, in both Figures, increases linearly by increasing the votes by factor of 10 on a logarithm scale, and what it means is that the execution time increases exponentially with increasing the votes by factor of 10. Indeed, it is the case, and both (extracted) codes, the unary natural number and Haskell native integer, are very slow, but the native integer code is marginally better than the unary natural number code.

4.5 Counting Millions of Ballots

The previous extracted Haskell code was very slow and was not practical for real life election involving millions of ballots. To scale it to real life election, we analysed the extracted Haskell code from Coq code. The most performance critical aspect of our code was the computation of margin function. Recall that the margin function is of type `cand -> cand -> Z` and that it depends on the *entire* set of ballots. Internally, it is represented by a closure [Landin, 1964] so that margins are re-computed with every call. The single largest efficiency improvement in our code was achieved by memoization, i.e. representing the margin function (in Coq) via list lookup. With this (and several smaller) optimisation, we can count millions of votes using verified code. However, this efficiency did not come for free, and we had to pay the cost in terms of (almost all) broken proofs. We had to redo all the proofs all over again[2]. Below (Figure 4.7, Figure 4.8), we include our timing graphs, based on randomly generated ballots while keeping number of candidates constant i.e.

[2]Redoing these proofs were trivial, but time consuming. I wished if there was a tool to automate this process

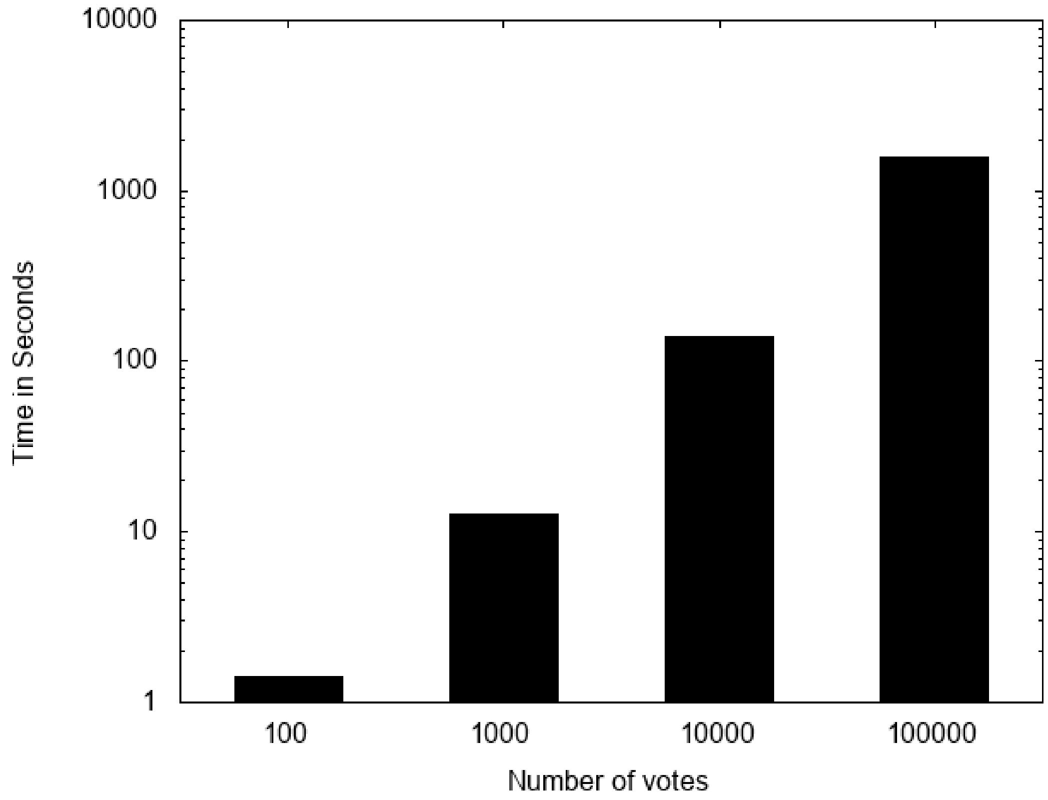

Figure 4.5: Experimental Result (Coq Unary Natural Number, Slow)

4 (The reason we kept it to 4 candidate to show the speed up compared to 4.5 and 4.6). During the experiment, we ran an election with 21 candidates, and we were able to count 2 million randomly generated ballots before running out of memory.)

In the Figure 4.7, we report timings (in seconds) for the computation of winners, whereas in the Figure 4.8, we include the time to additionally compute a universally verifiable certificate that attests to the correctness of the count. This is consistent with complexity of Schulze counting i.e. linear in number of ballots and cubic in number of candidates. The experiments were carried out on system equipped with intel core i7 processor and 16 GB of RAM. We notice that the computation of the certiciate adds comparatively little in computational cost.

Our implementation requires that we store *all* ballots in main memory as we need to parse the entire list of ballots before making it available to our verified implementation so that the total number of ballots we can count is

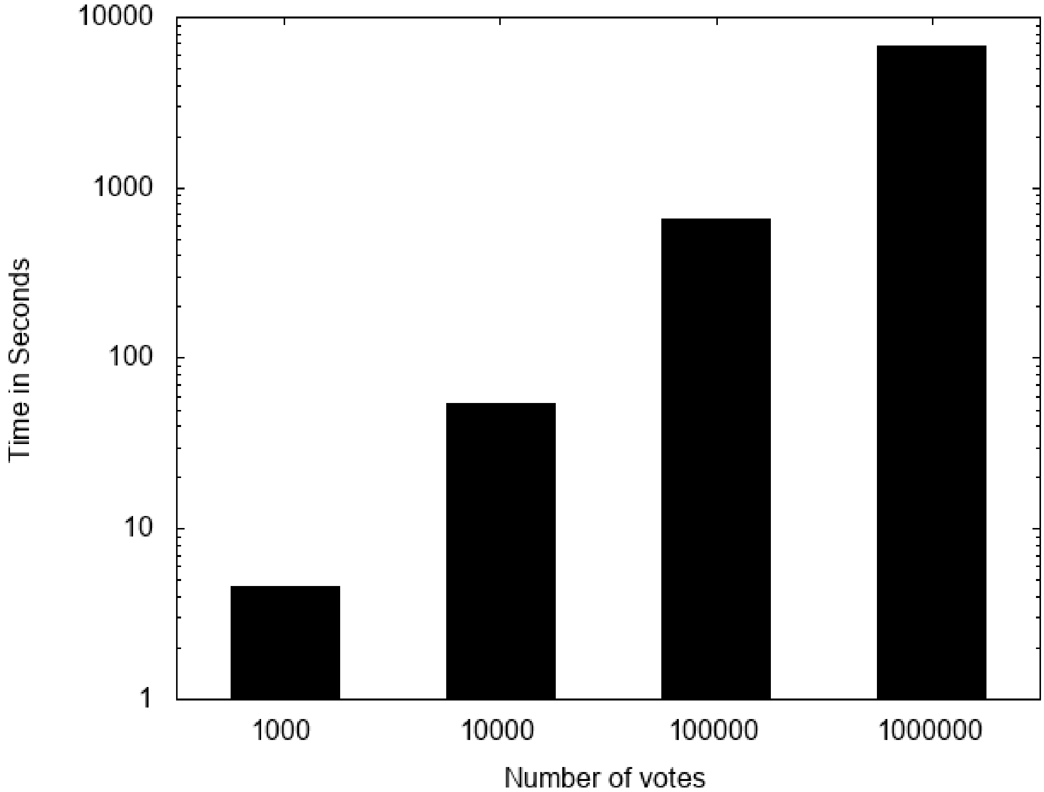

Figure 4.6: Experimental Result (Haskell Native Integer, Slow)

limited by main memory in practise. We can count real-world size elections (8 million ballot papers) on a standard, commodity desktop computer with 16 GB of main memory.

4.6 Discussion

In this chapter, we emphasize on correctness, and we take the approach that computation of winners in electronic voting (and in situations where correctness is key in general) should not only produce an end result, but an end result *together* with a verifiable justification of the correctness of the computed result. We have exemplified this approach by providing a provably correct, and evidence-producing implementation of vote counting according to the Schulze method.

While the Schulze method is not difficult to implement, and indeed there

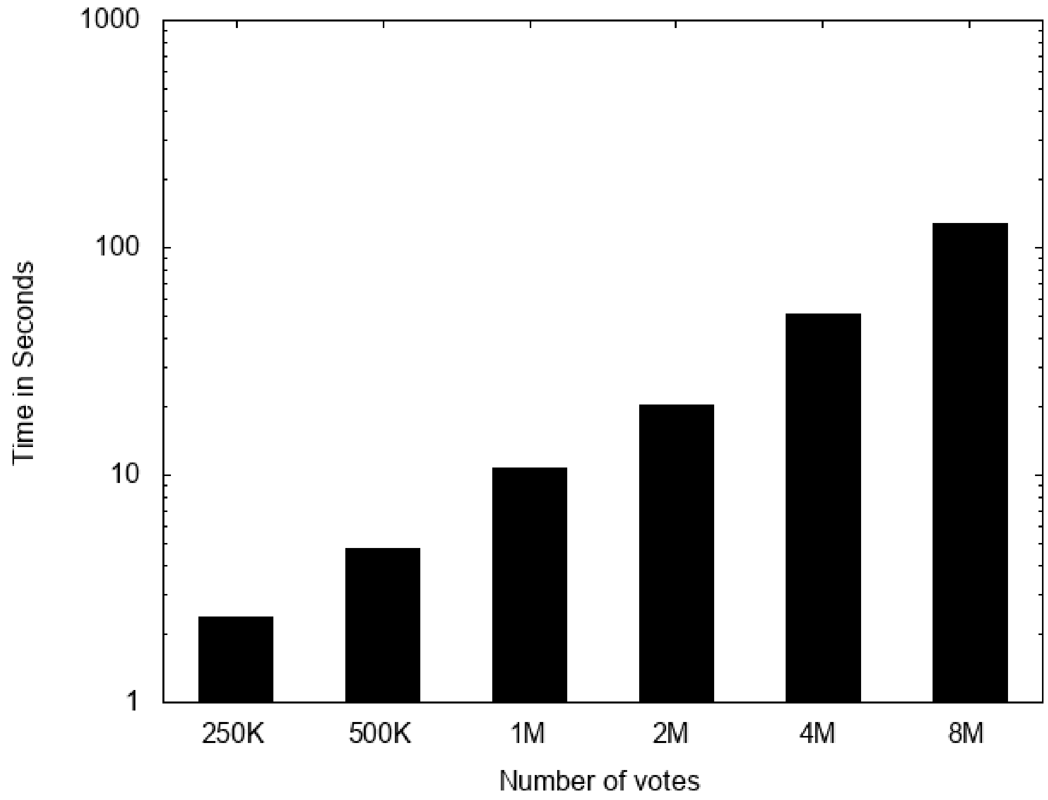

Figure 4.7: Computation of Winner (Without Certificate, Fast)

are many freely available implementations on the Internet, comparing the results between different implementations can give some level of assurance for correctness only in case the results agree. If there is a discrepancy, a certificate for the correctness of the count allows to adjudicate between different implementations, as the certificate can be checked with relatively little computational effort.

From the perspective of computational complexity, checking a transcript for correctness is of the same complexity as computing the set of winners, as our certificates are cubic in size, so that certificate checking is not less complex than the actual computation. However, publishing an independently verifiable certificate that attests the individual steps of the computation helps to increase *trust* in the computed election outcome. Typically, the use of technology in elections increases the amount of trust that we need to place both in technological artefacts, and in people. It raises questions that range from fundamental aspects, such as proper testing and/or verification of the software, to very practical questions, e.g. whether the correct version of the

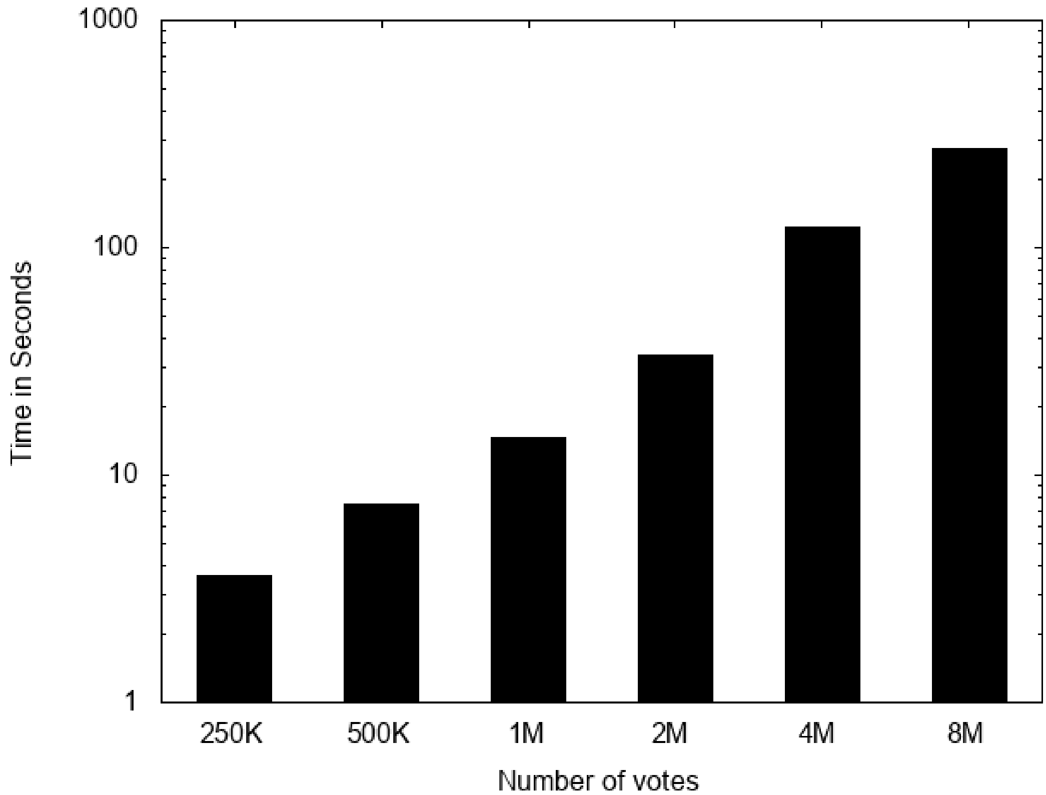

Figure 4.8: Computation of Winner (With Certificate, Fast)

software has been run. On the contrast, publishing a certificate of the count dramatically reduces the amount of trust that we need to place into both people and technology: the ability to publish a verifiable justification of the correctness of the count allows a large number of individuals to scrutinise the count. While only moderate programming skills are required to check the validity of a certificate (the transcript of the count), even individuals without any programming background can at least spot-check the transcript: for the construction of the margin function, everything that is needed is to show that the respective margins change according to the counted ballot. For the correctness of determination of winners, it is easy to verify existence of paths of a given strength, and also whether certain sets are coclosed – even by hand! This dramatically increases the class of people that can scrutinise the correctness of the count, and so helps to establish a trust basis that is much wider as no trust in software artefacts is required.

Technically, we do not *implement* an algorithm that counts votes according to the Schulze method. Instead, we give a specification of the Schulze winning conditions (`wins_prop` in Section 4.3) in terms of an already computed margin function that (we hope) can immediately be seen to be correct, and then show that those winning conditions are equivalent to the existence of inhabitants of types that carry verifiable evidence (`wins_type`). We then join the (type level) winning conditions with an inductive type that details the construction of the margin function in an inductive type. Via propositions-as-types, a provably correct vote counting function is then equivalent to the proposition that there exists an inhabitant of `Count` for every set of ballots. Coq's extraction mechanism then allows us to extract a Haskell program that produces election winners, together with verifiable certificates.

4.7 Summary

Our formalization achieves *Correctness*, *Practicality*, and (tallied-as-cast) *Verifiability*. The major problem in this formalization is *Privacy*. Our ballots are in plaintext and could easily be identified if the number of candidates participating in election in are large (Italian attack) [Otten, 2003]. In nutshell, the achieved and failed points of this formalization:

- Achieved
  - Correctness: The implementation is formalized in Coq with emphasis on generating evidence to convince everyone about the outcome of election.
  - Practicality: The extracted code can count millions of ballots. Therefore, we can use it in any real life election.
  - Verifiability: The outcome of any election can be verified by any third party using the generated certificates. Certificates generated for plaintext ballot during the election are very simple. It requires basic math literacy to audit the certificate which would lead to increase in number of scrutineers.
- Failed
  - Privacy: There is no privacy because the ballots involved are simply plaintext which could potentially lead coercion and vote-selling (coercion).

We remark that extracting Coq developments into a programming language itself is a non-verified process which could still introduce errors in our code. The most promising way to alleviate this is to independently implement (and verify) a certificate verifier, possibly in a language such as CakeML [Kumar et al., 2014] that is guaranteed to be correct to the machine level.

In the next chapter, we will try to solve privacy and coercion problem, using encryption, and to keep it verifiable, we will use zero-knowledge-proof. However, the solution for privacy comes at a cost, e.g. a loss in the pool of scrutineers because auditing a certificate generated by counting encrypted ballot requires intricate knowledge of cryptography.

Chapter 5

Homomorphic Schulze Algorithm : Axiomatic Approach

Be melting snow. Wash yourself of yourself.

Rumi

5.1 Introduction

As we stated in the summary of last chapter that plaintext could lead to privacy problems, e.g. ballot identification (Italian attack) Otten [2003]. In this chapter, we achieve privacy by using encryption, (tallied-as-cast) verifiability by using zero-knowledge proof, and correctness of implementation by proving the correctness properties inside the Coq theorem prover. One important point to note that we do not formalize any cryptographic primitive inside the Coq, but take an axiomatic approach, i.e. we assume the existence of cryptographic primitives and postulate their correctness property (axiomatisation of cryptographic primitives). The reason for axiomatic approach is because our goal is to not formalise the cryptographic primitives, but use these primitives to conduct an election which has all three ingredients, privacy, verifiability, and correctness. We then obtain, via program extraction, a provably correct implementation of vote counting, that we turn into executable code by providing implementations of the cryptographic primitives based on a standard cryptographic library (Unicrypt). We conclude by presenting experimental results, and discuss the trust base, security, and privacy as well as the applicability of our work to real-world scenarios.

Chapter Outline: In section 5.2, we discuss the technique to achieve verifiable homomorphic tally. In order to do so, we discuss why do we need our ballot to have a matrix representation and not a ranking function. Moreover, we discuss the concept of validity of a ballot, which comes naturally with matrix representation, steps of homomorphic counting, and cryptographic primitives needed to achieve all the required functionality. Section 5.3 takes a step forward and makes every concept from the previous section concrete using the Coq theorem prover. One important point in this section is our inductive data type *ECount* augmented with verification data in form of zero-knowledge proof for various claims made during the counting (*ECount* is conceptually similar to the *Count* (section 4.3.1), but in terms of data, *ECount* has state data related to counting and verification data in form of zero-knowledge proof, while *Count* has just state data). In section 5.4, we present our main theorem which states that for any set of given encrypted ballots, a winner can always be found. Apart from the main theorem, this section also incorporates the proof of correctness by stating the winners produced by encrypted ballots are same as plaintext ballots (section 4.3.2) if the encrypted ballots decrypt to the plaintext ballot. Section 5.5 focuses on extraction, instantiating the cryptographic primitives with UniCrypt library, and experimental results. Finally, section 5.6 highlights our assumptions, scalability issues, the goals we achieved and the goals we missed.

Secure elections are a balancing act between integrity and privacy: achieving either is trivial but their combination is notoriously hard. One of the key challenges faced by both paper based and electronic elections is that results must be substantiated with verifiable evidence of their correctness while retaining the secrecy of the individual ballot [Bernhard et al., 2017]. The combination of privacy and integrity can be realised using cryptographic techniques, where encrypted ballots (that the voters themselves cannot decrypt) are published on a bulletin board, and the votes are then processed, and the correctness of the final tally is substantiated, using homomorphic encryption [Hirt and Sako, 2000] and verifiable shuffling [Bayer and Groth, 2012]. Integrity can then be guaranteed by means of zero-knowledge proofs (ZKP), first studied by Goldwasser, Micali, and Rackoff [Goldwasser et al., 1985]. Informally, a zero-knowledge proof is a probabilistic and interactive proof where one entity interacts with another such that the interaction provides no information other than that the statement being proved is true with overwhelming probability. Later results [Ben-Or et al., 1988; Goldreich et al., 1991] showed that all problems for which solutions can be efficiently verified have zero-knowledge proof (in practice, Sigma protocol [Cramer et al., 1994] is used to prove the knowledge of a (private) witness w for a public input x, and it is required to be zero-knowledge against the honest verifier).

5.2 Verifiable Homomorphic Tallying

The realisation of verifiable homomorphic tallying that we are about to describe follows the same two phases as the Schulze algorithm (described in section 4.2): we first homomorphically compute the margin matrix from encrypted ballots, and then compute winners on the basis of the (decrypted) margin. Moreover, the computation also produces a verifiable certificate that leaks no information about choices in individual ballots other than the final tally, which in turn leaks no information about individual ballots if the number of voters is large enough.

5.2.1 Format of Ballots

Recall that in preferential voting schemes, ballots are rank-ordered lists of candidates. For the Schulze Method, we require that all candidates are ranked, and two candidates may be given the same rank. That is, a ballot is most naturally represented as a function, $C \to \mathbb{N}$, that assigns a numerical rank to each candidate, and the computation of the margin amounts to computing the sum

$$m(x,y) = \sum_{b \in B} \begin{cases} +1 & b(x) < b(y) \\ 0 & b(x) = b(y) \\ -1 & b(x) > b(y) \end{cases}$$

where B is the multi-set of ballots, and each $b \in B$ is a ranking function $b : C \to \mathbb{N}$ over a (finite) set C of candidates.

Ideally, we could have copied the same ballot structure in a homomorphic Schulze method, but encrypting the choices, i.e. the ballot would have been represented as a function $C \to CT$ where CT (ciphertext) is the encrypted representation of a choice (natural number). However, we note that this representation of ballots is not well suited for homomorphic computation of the margin matrix as practically feasible homomorphic encryption schemes do not support comparison operators and case distinctions as used in the formula above (to the best of our knowledge).

We instead represent ballots as candidate indexed matrices (represented as function from a pair of candidates to a natural number), $C \times C \to \mathbb{N}$, bm where $bm(x,y) = +1$ if x is preferred over y, $bm(x,y) = -1$ if y is preferred over x and $bm(x,y) = 0$ if x and y are equally preferred. The downside of this representation is that it takes $O(n^2)$ space to represent a ballot where n is the number of candidate participating in election.

While the advantage of the first representation, $b : C \to \mathbb{N}$, is that each ranking function is necessarily a valid ranking and is linear space ($O(n)$)) in the number of candidates, n, the advantage of the matrix representation, $C \times C \to \mathbb{N}$, is that the computation of the margin matrix is simple, that is

$$m(x,y) = \sum_{bm \in B} bm(x,y)$$

where bm is plaintext ballot in matrix form, and B is the multi-set of ballots (in matrix form). The major benefit of this matrix representation is that it can be transferred to the encrypted setting in a straight forward way by encrypting all the entries in bm. We take the advantage of matrix representation and represent our encrypted ballot as a matrix of ciphertext, $C \times C \to CT$ (CT stands for ciphertext). Following this representation of encrypted ballot, the encrypted margin can be computed as:

$$encm(x,y) = \bigoplus_{encb \in EncB} encb(x,y) \tag{5.1}$$

where $\oplus$ denotes homomorphic addition, $encb$ is an encrypted ballot in matrix form and $EncB$ is the multi-set of encrypted ballots.

In an ideal world where every voter is honest, every entry in $encb$ is either the encryption of -1, 0, or 1. Moreover, if we interpret $encb$ as a adjacency matrix of a graph representation, $encb$ should be acyclic (lets call it valid or desirable ballot). In addition, we definitely do not want to be in a situation where a voter prefers A over B, B over C, and finally C over A, for some candidates A, B, and C. How would we interpret a ballot having cycles? One possible interpretation is ranking all the candidates equal appearing a cycle, but clearly it is not a reasonable interpretation, therefore we decided to not include it in final tally (lets call it invalid or undesirable ballot). Now during the course of counting encrypted ballots, we need to distinguish between a desirable ballot, i.e. valid ballot, and an undesirable ballot, i.e. invalid ballot. To make this distinction, we verify that an encrypted (matrix) ballot $eballot : C \times C \to CT$ is valid only if:

- the decryption of $eballot(x,y)$ is indeed one of $1, 0$ or -1, where x and

y ranges over the list of all candidates

- *eballot* is acyclic (the idea here is that if *eballot* is acyclic, then for its decryption, $pballot : C \times C \to \mathbb{Z}$, we can find a ranking function from candidates to natural number, $C \to \mathbb{N}$, i.e. we can rank all the candidates in a linear order (explained in section 5.2.2)).

More importantly, to achieve verifiability, we not only need *verify* that a ballot is valid, we also need to *evidence* its validity (or otherwise) in the scrutiny sheet. However, the issue with the above definition, verification of the validity of an encrypted ballot, is that if we decrypt the *eballot* to the *pballot* and publish *pballot* in the scrutiny sheet to evidence the validity of *eballot*, we would reveal the (secret) preference of the voter, who cast this ballot. Therefore, we cannot decrypt the *eballot*, and we need a better way to evidence the validity of an encrypted ballot.

5.2.2 Validity of Ballots

A plaintext ballot $ptballot : C \times C \to \mathbb{Z}$ is *valid* if it is induced by a ranking function, i.e. there exists a function $f : C \to \mathbb{N}$ such that $ptballot(x, y) = 1$ if $f(x) < f(y)$, $ptballot(x, y) = 0$ if $f(x) = f(y)$ and $ptballot(x, y) = -1$ if $f(x) > f(y)$. In symbols:

$$ptballot(x,y) = \exists f : C \to \mathbb{N} \begin{cases} +1 & f(x) < f(y) \\ 0 & f(x) = f(y) \\ -1 & f(x) > f(y) \end{cases}$$

For a plaintext ballot, it is easy to decide if is valid or not valid. One crucial observation is that if *pballot* is valid, it also valid after permuting its each row and column with the same permutation, i.e. *ptballot* is valid if and only if $ptballot'$ is valid, where

$$ptballot'(x,y) = ptballot(\pi(x), \pi(y))$$

and $\pi : C \to C$ is a permutation of candidates. In fact, if $f : C \to \mathbb{N}$ is a ranking function for *ptballot*, $f \circ \pi$ is a ranking function for $ptballot'$. Also, notice that if *ptballot* is the decryption of $ctballot : C \times C \to CT$ and $ptballot'$

is the decryption of $ctballot' : C \times C \rightarrow CT$, the following holds:

$$ctballot'(x, y) = ctballot(\pi(x), \pi(y))$$

Using these ideas, we can evidence the validity of the encrypted ballot, $ctballot : C \times C \rightarrow CT$. The ballot *ctballot* is valid if and only if its decryption i.e. the plaintext ballot $ptballot(x, y) = decb(ctballot(x, y)$ is valid, where $decb : C \times C \rightarrow \mathbb{Z}$ denotes decryption function, and x and y ranges over the list of candidates. However, as we stated previously, we cannot publish the *ptballot* as an evidence of validity of the *ctballot* in the scrutiny sheet because then it will leak the voter's preference. Thus we generate a secret permutation, π, and subsequently, we publish $ctballot'$, row and column permuted ballot of *ctballot* by the secret permutation π, and its decryption $ptballot'$, row and column permutation of *ptballot* by the same secrete permutation π. Moreover, we augment the scrutiny sheet with the zero-knowledge proofs about various claim, e.g. we have decrypted the $ctballot'$ honestly and $ptballot'$ is indeed an honest decryption, the commitment of the secret permutation π, etc. In a nutshell, we can evidence the validity of a ciphertext ballot *ctballot* by

- generating a secret permutation, π and evidence that it is a valid permutation.
- publishing the shuffled version $ctballot'$ of *ctballot*, that is shuffled by the secret permutation π, together with evidence that $ctballot'$ is indeed a shuffle of *ctballot*.
- publishing the decryption $ptballot'$ of $ctballot'$ together with evidence that $ptballot'$ is indeed the decryption of $ctballot'$.
- evidencing if $ptballot'$ is valid (or otherwise) by showing the existence of ranking function $f : C \rightarrow \mathbb{N}$ (or otherwise).

We use zero-knowledge proofs (ZKP) in the style of [Terelius and Wikström, 2010] to evidence the correctness of the shuffle, and zero-knowledge proofs of honest decryption [Chaum and Pedersen, 1992] to evidence correctness of decryption. This achieves ballot secrecy as the (secret) permutation is never revealed.

In summary, the evidence of correct (homomorphic) counting starts with an encryption of the zero margin $encm : C \times C \rightarrow CT$ (every entry in the *encm* is an encrypted value of 0), and for each ciphertext ballot $ctballot : C \times C \rightarrow CT$ contains

1. generation of secret permutation π together with a commitment proof and zero-knowledge proof that it is a valid permutation
2. $ctballot'$, a shuffle of $ctballot$ by π together with a zero-knowledge proof that $ctballot'$ is a shuffle of $ctballot$
3. $ptballot'$, decryption of $ctballot'$, together with a zero-knowledge proof that $ptballot'$ is honest decryption of $ctballot'$
4. if $ptballot'$ is valid, we homorphically update the margin matrix $encm$, i.e.
$$encm(x, y) = encm(x, y) \bigoplus ctballot(x, y)$$
5. if the decrypted ballot $ptballot'$ is invalid, the margin matrix $encm$ remains unchanged

Once all ballots have been processed in this way, the certificate determines winners and contains winners by exhibiting

5. the final tally $encm$, together with its decryption and a zero-knowledge proof of honest decryption
6. publishes the winner(s), together with evidence to substantiate the claim (existence of strongest path from the winner to the loser and absence of such path from the loser to the winner, section 4.3).

5.2.3 Cryptographic primitives

We require an additively homomorphic cryptosystem to compute the (encrypted) margin matrix according to Equation 5.1 (this implements Item 4 above). All other primitives fall into one of three categories. *Verification primitives* are used to syntactically define the type of valid certificates. For example, when publishing the decrypted margin matrix in Item 5 above, we require that the zero-knowledge proof in fact evidences correct decryption. To guarantee this, we need a verification primitive that – given ciphertext, plaintext and zero-knowledge proof – verifies whether the supplied proof indeed evidences that the given ciphertext corresponds to the given plaintext. In particular, verification primitives are always boolean valued functions. While verification primitives *define* valid certificates, *generation primitives* are used to *produce* valid certificates. In the example above, we need a decryption primitive (to decrypt the homomorphically computed margin) and a primitive to generate a zero-knowledge proof (that witnesses correct decryption). Clearly verification and generation primitives have a close correlation, and we need to require, for example, that zero-knowledge proofs obtained via a generation primitive has to pass muster using the corresponding verification primitive.

The three primitives described above (decryption, generation of a zero-knowledge proof, and verification of this proof) already allow us to implement the entire protocol with exception of ballot shuffling (Item 2 above). Here, the situation is more complex. While existing mixing schemes (e.g. [Bayer and Groth, 2012]) permute an array of ciphertexts and produce a zero knolwedge proof that evidences the correctness of the shuffle, our requirement dictates that every row and colum of the (matrix) ballot is shuffled with the *same* (secret) permutation. In other words, we need to retain the identity of the permutation to guarantee that each row and column of a ballot have been shuffled by the same permutation. We achieve this by committing to a permutation using Pedersen's commitment scheme [Pedersen, 1992]. In a nutshell, the Pedersen commitment scheme has the following properties.

- Hiding: the commitment reveals no information about the permutation
- Binding: no party can open the commitment in more than one way, i.e. the commitment is to one permutation only.

A combination of Pedersen's commitment scheme with a zero-knowledge proof leads to a similar two step protocol, also known as commitment-consistent proof of shuffle [Wikström, 2009].

- Commit to a secret permutation and publish the commitment (hiding).
- Use a zero-knowledge proof to show that shuffling has used the same permutation which we committed to in previous step (binding).

This allows us to witness the validity (or otherwise) of a ballot by generating a permutation π which is used to shuffle every row and column of the ballot. We hide π by committing it using Pedersen's commitment scheme and record the commitment c_π in the certificate. However, for the binding step, rather than opening π we generate a zero-knowledge proof, zkp_π, using π and c_π, which can be used to prove that c_π is indeed the commitment to some permutation used in the (commitment consistent) shuffling without being opened [Wikström, 2009]. We can now use the permutation that we have committed to for shuffling each row and column of a ballot, and evidence the correctness of the shuffle via a zero-knowledge proof. To evidence validity (or otherwise) of a (single) ballot, we therefore:

1. generate a (secret) permutation and publish a commitment to this permutation, together with a zero-knowledge proof that evidences commitment to a permutation
2. for each row of the ballot, publish a shuffle of the row with the permutation committed to, together with a zero-knowledge proof that witnesses shuffle correctness
3. for each column of the row shuffled ballot, publish a shuffle of the column, also together with a zero-knowledge proof of correctness
4. publish the decryption the ballot shuffled in this way, together with a zero-knowledge proof that witnesses honest decryption
5. decide the validity of the ballot based on the decrypted shuffle.

The cryptographic primitives needed to implement this again fall into the same classes. To define validity of certificates, we need verification primitives

- to decide whether a zero-knowledge proof evidences that a given commitment indeed commits to a permutation
- to decide whether a zero-knowledge proof evidences the correctness of a shuffle relative to a given permutation commitment.

Dual to the above, to generate (valid) certificates, we need the ability to

- generate permutation commitments and accompanying zero-knowledge proofs that evidence commitment to this permutation
- generate shuffles relative to a commitment, and zero-knowledge proofs that evidence the correctness of shuffles.

Again, both need to be coherent in the sense that the zero-knowledge proofs produced by the generation primitives need to pass validation. In summary, we require an additively homomorphic cryptosystem that implements the following:

Decryption Primitives. decryption of a ciphertext, creation and verification of honest decryption zero-knowledge proofs.

Commitment Primitives. generating permutations, creation and verification of commitment zero-knowledge proofs

Shuffling Primitives. commitment consistent shuffling, creation and verification of commitment consistent zero-knowledge shuffle proofs

5.2.4 Witnessing of Winners

Once all ballots are counted, the computed margin is decrypted, and winners (together with evidence of winning) are computed using plaintext counting. We discuss this part only briefly, for sake of completeness, as it is identical to section 4.3. For each of the winners w and each candidate x we publish

- a natural number k and a path $x_0, \ldots, x_n$ of strength k, where $x_0 = w$ and $x_n = x$
- a set $C(w, x)$ of pairs of candidates that is k-coclosed and contains (x, w)

where a set S is k-coclosed if for all $(x, z) \in C$ we have that $m(x, z) < k$ and either $m(x, y) < k$ or $(y, z) \in S$ for all candidates y. Informally, the first requirement ensures that there is no direct path (of length one) between a pair $(x, z) \in S$, and the second requirement ensures that for an element $(x, z) \in S$, there cannot be a path that connects x to an intermediate node y and then (transitively) to z that is of strength $\geq k$.

5.3 Formalization in Coq

As we stated in the beginning of this chapter that the purpose of this work is not to verify cryptographic primitives, but use them as a tool to construct evidence which can be used to audit and verify the outcome during different phase of election. Here, we treat them as abstract entities and assume axioms about them inside Coq. In particular, we assume the existence of functions that implement each of the primitives described in the previous section, and postulate natural axioms that describe how the different primitives interact. As a by-product, we obtain an axiomatisation of a cryptographic library that we could, in a later step, verify the implementation of a cryptosystem against. In particular, this allows us to not commit to any particular cryptosystem in particular (although our development, and later instantiation, is geared towards ElGamal [ElGamal, 1985]).

The first part of our formalisation concerns the cryptographic primitives that we collect in a separate module. Below is an example of the generation / verification primitives for decryption, together with coherence axioms.

```
Variable decrypt_message:
  Group -> Prikey -> ciphertext ->  plaintext.

Variable construct_zero_knowledge_decryption_proof:
  Group -> Prikey -> ciphertext -> DecZkp.

Axiom verify_zero_knowledge_decryption_proof:
  Group -> plaintext -> ciphertext -> DecZkp -> bool.

Axiom honest_decryption_from_zkp_proof: forall group c d zkp,
 verify_zero_knowledge_decryption_proof group d c zkp = true
 -> d = decrypt_message grp privatekey c.

Axiom verify_honest_decryption_zkp (group: Group):
  forall (pt : plaintext) (ct : ciphertext) (pk : Prikey),
  (pt = decrypt_message group pk ct) ->
  verify_zero_knowledge_decryption_proof group pt ct
  (construct_zero_knowledge_decryption_proof group pk ct)
  = true.
```

The different keywords `Variable` and `Axiom` are used as a convenience for extraction. The keyword `Variable` is used if we want it to be lambda abstracted otherwise keyword `Axiom`. In the above, the first two functions, `decrypt_message` and `construct_zero _knowledge _decryption _proof` are *generation* primitives, whereas the function `verify_zero _knowledge` `_decryption _proof` is a *verification* primitive. We have two coherence axioms. The first says that if the verification of a zero-knowledge proof of honest decryption succeeds, then the ciphertext indeed decrypts to the given plaintext. The second stipulates that generated zero-knowledge proofs indeed verify.

For ballots, we assume a type `cand` of candidates, and represent plaintext and encrypted ballots as two-argument functions that take plaintext, and ciphertexts, as values.

```
Definition pballot := cand -> cand -> plaintext.
Definition eballot := cand -> cand -> ciphertext.
```

We now turn to the representation of certificates, and indeed to the definition of what it means to (a) count encrypted votes correctly according to the Schulze Method, and (b) produce a verifiable certificate of this fact. At a high level, we split the counting (and accordingly the certificate) into *states*. This gives rise to a (inductive dependent) type `ECount`, parameterised by the ballots being counted.

```
Inductive ECount (group : Group) (bs : list eballot) :
  EState -> Type
```

Given a list `bs` of ballots, `ECount bs` is a inductive dependent type. In this case, given a state of counting (i.e. an inhabitant `estate` of `EState`), the type level application `ECount bs estate` is the *type of evidence that proves that* `estate` *is a state of counting that has been reached according to the method*. The states itself are represented by the type `EState` where

- `epartial` represents a partial state of counting, consisting of the homomorphically computed margin so far, the list of uncounted ballots and the list of invalid ballots encountered so far
- `edecrypt` represents the final decrypted margin matrix, and
- `ewinners` is the final determination of winners.

This is readily translated to the following Coq code:

```
Inductive EState : Type :=
 | epartial : (list eballot * list eballot) ->
              (cand -> cand -> ciphertext) -> EState
 | edecrypt : (cand -> cand -> plaintext) -> EState
 | ewinners : (cand -> bool) -> EState.
```

The constructors of `EState` then allow us to move from one state to the next, under appropriate conditions that guarantee correctness of the count. The different states during the counting represented by *ECount* is tagged by five constructors:

- `ecax`: marks the beginning of counting
- `ecvalid`: process a ballot from cast-ballots pile, and the ballot is a valid ballot
- `ecinvalid`: process a ballot from cast-ballot pile, and the ballot is a invalid ballot
- `ecdecrypt`: decryption of fully constructed homomorphic margin from the cast-ballot
- `ecfin`: declaration of winner and loser based on the decrypted margin

Inductive type *ECount* with all the constructors filled with *state data*, *verification data*, and *correctness constraint*. The first constructor, `ecax`, bootstraps the count and ensures that

- all ballots are initially uncounted
- margin matrix is an encryption of the zero matrix

state data: here, the list of uncounted and invalid ballots, and the encrypted homomorphic margin

verification data: a zero-knowledge proof that the encrypted homomorphic margin is indeed an encryption of the zero margin

correctness constraints: here, the constructor may only be applied if the list of uncounted ballots is equal to the list of ballots cast, and the fact that the zero-knowledge proofs indeed verify that the initial margin matrix is identically zero.

The main difference between the correctness condition, and the verification data is that the former can be simply be inspected (here by comparing lists) whereas the latter requires additional data (here in the form of a zero-knowledge proof). In Coq, this constructor can be encoded as:

```
Inductive ECount (grp : Group) (bs : list eballot) :
 EState -> Type :=
 | ecax (us : list eballot)
     (encm : cand -> cand -> ciphertext)
     (decm : cand -> cand -> plaintext)
     (zkpdec : cand -> cand -> DecZkp) :
     us = bs ->
     (forall c d : cand, decm c d = 0) ->
     (forall c d, verify_zero_knowledge_decryption_proof
         grp (decm c d) (encm c d) (zkpdec c d) = true) ->
   ECount grp bs (epartial (us, []) encm)
```

The constructor `ecvalid` represents the effect of counting a valid ballot. Here the crucial aspect is that validity needs to be evidenced. As before, we have:

state data: as before, the list of uncounted and invalid ballots, the homomorphic margin, but additionally evidence that the previous state has been obtained correctly

verification data: a commitment to a (secret) permutation, a row permutation of the ballot being counted, and a column permutation of this, and a decryption of the row- and column permuted ballot (all with accompanying zero-knowledge proofs)

correctness constraints: all the zero-knowledge proofs verify, the new margin is the homomorphic addition of the previous margin and the counted ballot, and the decrypted (shuffled) ballot is indeed valid.

```
 | ecvalid (u : eballot) (v : eballot) (w : eballot)
    (b : pballot) (zkppermuv : cand -> ShuffleZkp)
    (zkppermvw : cand -> ShuffleZkp)
    (zkpdecw : cand -> cand -> DecZkp)
    (cpi : Commitment) (zkpcpi : PermZkp)
    (us : list eballot)
```

```
(m nm : cand -> cand -> ciphertext)
(inbs : list eballot) :
ECount grp bs (epartial (u :: us, inbs) m) ->
(* valid ballot *)
matrix_ballot_valid b ->
(* commitment proof *)
verify_permutation_commitment grp
    (List.length cand_all) cpi zkpcpi = true ->
(forall c, verify_row_permutation_ballot grp
    u v cpi zkppermuv c = true)  ->
(forall c, verify_col_permutation_ballot grp
    v w cpi zkppermvw c = true)  ->
(forall c d, verify_zero_knowledge_decryption_proof
     grp (b c d) (w c d) (zkpdecw c d) = true)  ->
(forall c d, nm c d = homomorphic_addition
     grp (u c d) (m c d)) ->
ECount grp bs (epartial (us, inbs) nm)
```

The constructor `ecinvalid` is very similar to `ecvalid`. We elide the description of the constructor that is applied when an invalid ballot is being encountered (the only difference is that the margin matrix is not being updated and ballot is moved to list of invalid ballots).

```
| ecinvalid (u : eballot) (v : eballot) (w : eballot)
   (b : pballot) (zkppermuv : cand -> ShuffleZkp)
   (zkppermvw : cand -> ShuffleZkp)
   (zkpdecw : cand -> cand -> DecZkp)
   (cpi : Commitment) (zkpcpi : PermZkp)
   (us : list eballot) (m : cand -> cand -> ciphertext)
   (inbs : list eballot) :
   ECount grp bs (epartial (u :: us, inbs) m) ->
   (* invalid ballot *)
   ~matrix_ballot_valid b ->
   (* commitment proof *)
   verify_permutation_commitment grp
     (List.length cand_all) cpi zkpcpi = true ->
   (forall c, verify_row_permutation_ballot grp
     u v cpi zkppermuv c = true) ->
   (forall c, verify_col_permutation_ballot grp
     v w cpi zkppermvw c = true) ->
```

```
    (forall c d, verify_zero_knowledge_decryption_proof
       grp (b c d) (w c d) (zkpdecw c d) = true) ->
    ECount grp bs (epartial (us, (u :: inbs)) m)
```

Counting finishes when there are no more uncounted ballots and this state is marked by constructor *ecdecrypt*, in which case the next step is to publish the decrypted margin matrix. Also here, we have

state data: the decrypted margin matrix, plus evidence that a state with no more uncounted ballots has been obtained correctly

verification data: a zero-knowledge proof that demonstrates honest decryption of the final margin matrix

correctness constraints: the given zero-knowledge proof verifies, i.e. the given decrypted margin is indeed the decryption of the (last) homomorphically computed margin matrix.

```
  | ecdecrypt inbs
      (encm : cand -> cand -> ciphertext)
      (decm : cand -> cand -> plaintext)
      (zkp : cand -> cand -> DecZkp) :
      ECount grp bs (epartial ([], inbs) encm) ->
      (forall c d, verify_zero_knowledge_decryption_proof
         grp (decm c d) (encm c d) (zkp c d) = true) ->
      ECount grp bs (edecrypt decm)
```

The last constructor, *ecfin*, finally declares the winners of the election, and we have:

state data: a function `cand -> bool` that determines winners, plus evidence of the fact that the decrypted final margin matrix has been obtained correctly

verification data: paths and co-closed sets that evidence the correctness of the function above

correctness constraints: that ensure that the verification data verifies the winners given by the state data.

This last part is same as the previous chapter's scrutiny sheet (section 4.4).

```
  | ecfin dm w
     (d : (forall c, (wins_type dm c) +
                     (loses_type dm c))) :
     ECount grp bs (edecrypt dm) ->
     (forall c, w c = true <-> (exists x, d c = inl x)) ->
     (forall c, w c = false <-> (exists x, d c = inr x)) ->
     ECount grp bs (ewinners w).
```

5.4 Correctness by Construction and Verification

In the previous section, we have presented a data type that *defines* the notion of a verifiably correct count of the Schulze Method, on the basis of encrypted ballots. To obtain an executable that in fact *produces* a verifiable (and provably correct) count, we can proceed in either of two ways:

1. implement a function that – given a list `bs` of ballots – produces a boolean function `w` (for winners) and an element of the type `ECount bs (winners w)`. This gives both the election winners (`w`) as well as evidence (the element of the `ECount` data type).

2. to prove that for every set `bs` of encrypted ballots, we have a boolean function `w` and an inhabitant of the type `ECount bs (winners w)`.

Under the proofs-as-programs interpretation of constructive type theory, both amount to the same. We chose the latter approach, and our main theorem formally states that all elections can be counted according to the Schulze Method (with encrypted ballots), i.e. a winner can always be found. Formally, our main theorem takes the following form:

```
Lemma encryption_schulze_winners (group : Group)
 (bs : list eballot) : existsT (f : cand -> bool),
 ECount group bs (ewinners f).
```

The proof proceeds by successively building an inhabitant of `EState` by homomorphically computing the margin matrix, then decrypting and determining the winners. Within the proof, we use both generation primitives (e.g. to construct zero-knowledge proofs) and coherence axioms (to ensure that the zero-knowledge proofs indeed verify).

The correctness of our entire approach stands or falls with the correct formalisation of the inductive data type `ECount` that is used to determine the winners of an election counted according to the Schulze Method. While one can argue that the data type itself is transparent enough to be its own specification, the cryptographic aspect makes things slightly more complex. For example, it appears to be credible that our mechanism for determining validity of a ballot is correct – however we have not given proof of this. Rather than scrutinising the details of the construction of this data type, we follow a different approach: we demonstrate that homomorphic counting always yields the same results as plaintext counting, where plaintext counting is already verified against its specification (Chapter 4). This correspondence has two directions, and both assume that we are given two lists of ballots that are the encryption (resp. decryption) of one another.

The first theorem, `plaintext_ schulze_to _homomorphic`, reproduced below shows that every winner that can be determined using plaintext counting can also be evidenced on the basis of corresponding encrypted ballots. The converse of this is established by Theorem `homomorphic _schulze _to_plaintext`.

```
Lemma plaintext_schulze_to_homomorphic
 (group : Group) (bs : list ballot):
 forall (pbs : list pballot) (ebs : list eballot)
 (w : cand -> bool), (pbs = map (fun x => (fun c d =>
 decrypt_message group privatekey (x c d))) ebs) ->
 (mapping_ballot_pballot bs pbs) ->
 Count bs (winners w) -> ECount group ebs (ewinners w).

Lemma homomorphic_schulze_to_plaintext
 (group : Group) (bs : list ballot):
 forall (pbs : list pballot) (ebs : list eballot)
 (w : cand -> bool) (pbs = map (fun x => (fun c d =>
 decrypt_message group privatekey (x c d))) ebs) ->
 (mapping_ballot_pballot bs pbs) ->
 ECount grp ebs (ewinners w) -> Count bs (winners w).
```

The theorems above feature a third type of ballot that is the basis of plaintext counting, and is a simple ranking function of type `cand -> Nat`, and the two hypotheses on the three types of ballots ensure that the encrypted ballots (`ebs`) are in fact in alignment with the rank-ordered ballots (`bs`) that are used in plaintext counting. The proof, and indeed the formulation, relies on an inductive data type `Count` (Section 4.3.1) that can best be thought of as a plaintext version of the inductive type `ECount` given here. Crucially, `Count` is verified against a formal specification of the Schulze Method. Both theorems are proven by induction on the definition of the respective data types, where the key step is to show that the (decrypted) final margins agree. The key ingredient here are the coherence axioms that stipulate that zero-knowledge proofs that verify indeed evidence shuffle and/or honest decryption.

5.5 Extraction and Experiments

As discussed in section 3.1.3, we are using the Coq extraction mechanism [Letouzey, 2003] to extract programs from existence proofs[1]. In particular, we extract the proof of the Theorem `pschulze_winners`, given in section 5.4 to a program that delivers not only provably correct counts, but also verifiable evidence. Give a set of encrypted ballots and a `Group` that forms the basis of cryptographic operations, we obtain a program that delivers not only a set of winners, but additionally independently verifiable evidence of the correctness of the count.

Indeed, the entire formulation of our data type, and the split into state data, verification data, and correctness constraints, has been geared towards extraction as a goal. Technically, the verification conditions are *propositions*, i.e. inhabitants of Type *Prop* in the terminology of Coq, and hence erased at extraction time. This corresponds to the fact that the assertions embodied in the correctness constraints can be verified with minimal computational overhead, given the state and the verification data. For example, it can simply be verified whether or not a zero-knowledge proof indeed verifies honest decryption by running it through a verifier. On the other hand, the zero-knowledge proof itself (which is part of the verification data) is crucially needed to be able to verify that a plaintext is the honest decryption of a ciphertext, and hence cannot be erased during extraction. Technically, this is realised by formulating both state and verification data at type level (rather than as propositions).

As we have explained in section 5.3, the formal development does not pre-suppose any specific implementation of the cryptographic primitives, and we assume the existence of cryptographic infrastructure. From the perspec-

[1] https://github.com/mukeshtiwari/EncryptionSchulze/tree/master/code/Workingcode

tive of extraction, this produces an executable with "holes", i.e. the cryptographic primitives need to be supplied to fill the holes and indeed be able to compile and execute the extracted program.

To fill this hole, we implement the cryptoraphic primitives with help of the UniCrypt library [Locher and Haenni, 2014]. UniCrypt is a freely available library, written in Java, that provides nearly all of the required functionalities, with the exception of honest decryption zero-knowledge proofs. We extract our proof development into OCaml and use Java/OCaml bindings [Aguillon] to make the UniCrypt functionality available to our OCaml program. After instantiating the cryptographic primitives in the extracted OCaml code with wrapper code that calls UniCrypt, we tested the executable on a three candidate elections between candidates `A`, `B` and `C`. The computation produces a tally sheet that is schematically given below: it is trace of computation which can be used as a checkable record to verify the outcome of election. We elide the cryptographic detail, e.g. the concrete representation of zero-knowledge proofs. A certificate is be obtained from the type `ECount` where the head of the certificate corresponds to the base case of the inductive type, here `ecax`. Below, `M` is encrypted margin matrix, `D` is its decrypted equivalent, required to be identically zero, and `Z` represents a matrix of zero knwoledge proofs, each establishing that the `XY`-component of `M` is in fact an encryption of zero. All these matrices are indexed by candidates and we display these matrices by listing their entries prefixed by a pair of candidates, e.g. the ellipsis in `AB(...)` denotes the matrix entry at row `A` and column `B`.

```
M: AB(rel-marg-of-A-over-B-enc), AC(rel-marg-of-A-over-C-enc), ...
D: AB(0)                       , AC(0)                       , ...
Z: AB(zkp-for-rel-marg-A-B)    , AC(zkp-for-rel-marg-A-C)    , ...
```

Note that one can verify the fact that the initial encrypted margin is in fact the zero margin by just verifying the zero-knowledge proofs. Successive entries in the certificate will generally be obtained by counting valid, and discarding invalid ballots. If a valid ballot is counted after the counting commences, the certificate would continue by exhibiting the state and verification data contained in the `ecvalid` constructor which can be displayed schematically as follows:

```
V: AB(ballot-entry-A-B) , AC(ballot-entry-A-C), ...
C: permutation-commitment
P: zkp-of-valid-permutation-commitment
```

```
R: AB(row-perm-A-B)     , AC(row-perm-A-C)    , ...
RP: A(zkp-of-perm-row-A), B(zkp-of-perm-row-B), ...
C: AB(col-perm-A-B),      AC(col-perm-A-C)    , ...
CP: A(zkp-of-perm-col-A), B(zkp-of-perm-col-B), ...
D: AB(dec-perm-bal-A-B) , AC(dec-perm-bal-A-C), ...
Z: AB(zkp-for-dec-A-B)  , AC(zkp-for-dec-A-C) , ...
M: AB(new-marg-A-B)     , AC(new-marg-A-C)    , ...
```

Here `V` is the list of ballots to be counted, where we only diplay the first element. We commit to a permutation and validate this commitment with a zero-knowledge proof, here given in the second and third line, prefixed with `C` and `P`. The following two lines are a row permutation of the ballot `V`, together with a zero-knowledge proof of correctness of shuffling (of each row) with respect to the permutation committed to by `C` above. The following two lines achieve the same for subsequently permuting the columns of the (row permuted) ballot. Finally, `D` is the decrypted permuted ballot, and `Z` a zero-knowledge proof of honest decryption. We end with an updated homomorphic margin matrix `M`. Again, we note that the validity of the decrypted ballot can be checked easily, and validating zero-knowledge proofs substantiate that the decrypted ballot is indeed a shuffle of the original one. Homomorphic addition can simply be re-computed.

The steps where invalid ballots are being detected are similar, with the exception of not updating the margin matrix. Once all ballots are counted, the only applicable constructor is `ecdecrypt`, the data content of which would continue a certificate schematically as follows:

```
V: []
M: AB(fin-marg-A-B), AC(fin-marg-A-C), ...
D: AB(dec-marg-A-B), AC(dec-marg-A-C), ...
Z: AB(zkp-dec-A-B) , AC(zkp-dec-A-C) , ...
```

Here the first line indicates that there are no more ballots to be counted, `M` is the final encrypted margin matrix, `D` is its decryption and `Z` is a matrix of zero-knowledge proofs verifying the correctness of decryption.

The certificate would end with the determination of winners based on the encrypted margin, and would end with the content of the `ecfin` constructor

```
winning: A, <evidence that A wins against B and C>
```

```
losing:  B, <evidence that B loses against A and C>
losing:  C, <evidence that C loses against A and B>
```

where the notion of evidence for winning and losing is as in the plaintext version of the protocol (Chapter 4).

Concrete Certificate: Below is a glimpse of a concrete certificate for an election. We have stripped off the trailing digits in the tally sheet which is marked by .., and rather than representing an entry of a matrix as (i, j), it is represented as ij

```
M: AA(13.., 10..) AB(90.., 14..)  AC(11.., 23..) BA(16.., 13..)
BB(79.., 46..) BC(12.., 14..) CA(50.., 53..) CB(70.., 68..) CC(23.., 82..),
D: [AA: 0 AB: 0 AC: 0 BA: 0 BB: 0 BC: 0 CA: 0 CB: 0 CC: 0],
Zero-Knowledge-Proof-of-Honest-Decryption: [..]
---------------------------------------------------------------------------
V: [AA(42.., 15..) AB(63.., 32..) AC(70, 44..) BA(47.., 34..) BB(16.., 28..)
BC(39.., 16..) CA(19.., 13..) CB(57.., 12..) CC(19.., 89..),..], I:  [],
M: AA(12.., 11..) AB(13.., 66..) AC(16.., 14.) BA(48.., 31..) BB(15.., 52..)
BC(15.., 68..) CA(39.., 69..) CB(12.., 78..) CC(10.., 40..),
Row-Permuted-Ballot: AA(53.., 16..) AB(23.., 44..) AC(72.., 47..)
BA(10.., 19..) BB(74.., 16..) BC(20.., 60..) CA(44.., 10..) CB(12.., 16..)
CC(59.., 98..),
Column-Permuted-Ballot: AA(81.., 41..) AB(17.., 14..) AC(10.., 14..)
BA(37.., 12..) BB(14.., 66..) BC(10.., 13..) CA(12.., 13..) CB(14.., 16..)
CC(12.., 10..),
Decryption-of-Permuted Ballot: AA0 AB-1 AC1 BA1 BB0 BC1 CA-1 CB-1 CC0,
Zero-Knowledge-Proof-of-Row-Permutation: [Tuple[...]],
Zero-Knowledge-Proof-of-Column-Permutation: [Tuple[..]],
Zero-Knowledge-Proof-of-Decryption: [Triple[..]],
Permutation-Commitment: Triple[..]
Zero-Knowledge-Proof-of-Commitment: Tuple[..]
---------------------------------------------------------------------------
.
.
.
---------------------------------------------------------------------------
V: [AA(36.., 10..) AB(20.., 13..) AC(75.., 43..) BA(13.., 31..) BB(27.., 82..)
BC(31.., 50..) CA(16.., 11..) CB(74.., 15..) CC(26.., 36..)], I: [],
M: AA(86.., 38..) AB(21.., 14..) AC(16.., 25..) BA(16.., 22..) BB(18.., 15..)
BC(11.., 63..) CA(15.., 34..) CB(76.., 18..) CC(11.., 10..),
Row-Permuted-Ballot: .., Column-Permuted-Ballot: ..,
Decryption-of-Permuted-Ballot: AA0 AB-10 AC1 BA10 BB0 BC1 CA-1 CB-1 CC0,
Zero-Knowledge-Proof-of-Row-Permutation: [..],
Zero-Knowledge-Proof-of-Column-Permutation: [..],
Zero-Knowledge-Proof-of-Decryption: [..],
Permutation-Commitment: Triple[..],
Zero-Knowledge-Proof-of-Commitment: Tuple[..]
---------------------------------------------------------------------------
V: [], I: [AA(36.., 10..) AB(20.., 13..) AC(75.., 43..) BA(13.., 31..)
BB(27.., 82..) BC(31.., 50..) CA(16.., 11..) CB(74.., 15..) CC(26.., 36..)],
M: .., D: [AA: 0 AB: 4 AC: 4 BA: -4 BB: 0 BC: 4 CA: -4 CB: -4 CC: 0],
Zero-Knowledge-Proof-of-Decryption: [..]
---------------------------------------------------------------------------
D: [AA: 0 AB: 4 AC: 4 BA: -4 BB: 0 BC: 4 CA: -4 CB: -4 CC: 0]
winning: A
   for B: path A --> B of strength 4, 5-coclosed set:
```

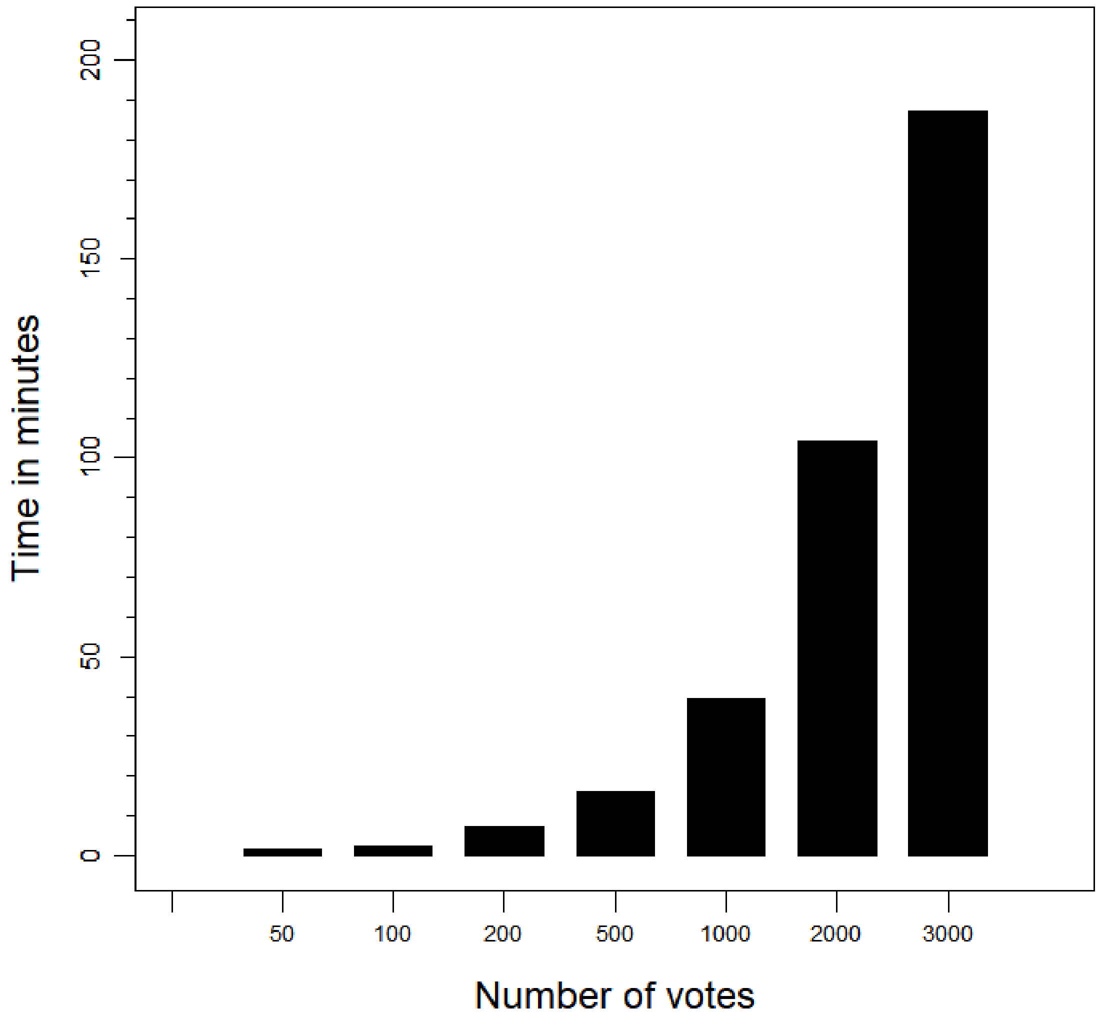

Figure 5.1: Experimental Result

```
      [(B,A),(C,A),(C,B)]
   for C: path A --> C of strength 4, 5-coclosed set:
      [(B,A),(C,A),(C,B)]
losing: B
   exists A: path A --> B of strength 4, 4-coclosed set:
     [(A,A),(B,A),(B,B),(C,A),(C,B),(C,C)]
losing: C
   exists A: path A --> C of strength 4, 4-coclosed set:
     [(A,A),(B,A),(B,B),(C,A),(C,B),(C,C)]
```

We note that the schematic presentation of the certificate above is nothing but a representation of the data contained in the extracted type `ECount` that we have chosen to present schematically. Concrete certificates can be inspected with the accompanying proof development, and are obtained by simply implementing datatype to string conversion on the type `ECount`.

To demonstrate proof of concept, we have run our experiment on an Intel i7 2.6 GHz Linux desktop computer with 16GB of RAM for three candidates and randomly generated ballots (Figure 5.1). The largest amount of ballot we counted was 10,000 (not included in graph), with a runtime of 25 hours. A more detailed analysis reveals that the bottleneck are the bindings between OCaml and Java. More specifically, producing the cryptographic evidence using the UniCrypt Library for 10,000 ballots takes about 10 minutes, and the subsequent computation (which is the same as for the plaintext count) takes negligible time. This is consistent with the mechanism employed by the bindings: each function call from OCaml to Java is inherently memory bounded and creates an instance of the Java runtime, the conversion of OCaml data structures into Java data structures, computation by respective Java function producing result, converting the result back into OCaml data structure, and finally destroying the Java runtime instance when the function returns. While the proof of concept using OCaml/Java bindings falls short of being practically feasible, our timing analysis substantiates that feasibility can be achieved by eliminating the overhead of the bindings.

5.6 Summary

The main contribution of our formalisation is that of independently verifiable *evidence* for a set of candidates to be the winners of an election counted according to the Schulze method. Our main claim is that our notion of evidence is both safeguarding the privacy of the individual ballot (as the count is based on encrypted ballots) and is verifiable at the same time (by means of zero knowledge proofs). To do this, we have axiomatised a set of cryptographic primitives to deal with encryption, decryption, correctness of shuffles and correctness of decryption. From formal and constructive proof of the fact that such evidence can always be obtained, we have then extracted executable code that is provably correct by construction and produces election winners together with evidence once implementations for the cryptographic primitives are supplied.

In a second step, we have supplied an implementation of these primitives, largely based on the UniCrypt Library. Our experiments have demonstrated that this approach is feasible, but quite clearly much work is still needed to improve efficiency.

Assumptions for Provable Correctness. While we claim that the end product embodies a high level of reliability, our approach necessarily leaves some gaps between the executable and the formal proofs. First and foremost, this is of course the implementation of the cryptographic primitives in an external (and unverified) library. We have minimised this gap by basing our implementation on a purpose-specific existing library (UniCrypt) to which we relegate most of the functionality.

Modelling Assumptions. In our modelling of the cryptographic primitives, in particular the zero-knowledge proofs, we assumed properties which in reality only hold with very high probability. As a consequence our correctness assertions only hold to the level of probability that is guaranteed by zero-knowledge proofs (Sigma protocols).

Scalability. We have analysed the feasibility of the extracted code by counting an increasing number of ballots. While this demonstrates a proof of concept, our results show that the bindings used to couple the cryptographic layer with our code adds significant overhead compared to plaintext tallying in Schulze method (4). Given that both parts are practically efficient by themselves, scalability is merely the question of engineering a more efficient coupling.

In a nutshell, the achieved and failed parts of this formalization:

- Achieved
  - Correctness: The implementation is formalized in Coq assuming the existence of cryptographic functions and axioms about their correctness behaviour. These primitives were used for constructing evidence, or certificate.
  - Privacy : We don't reveal the content of ballot at any phase of election counting. Therefore, there is no possibility of anyone knowing the choices of a voter other than the voter herself.
  - Verifiability: The outcome of any election can be verified by any third party because of the generated certificates. However, the nature of certificates in this case is very complex and can only be scrutinize by someone having specialized knowledge of cryptography which decreases the pool of potential scrutinizers dramatically.
- Failed
  - Correctness: We use an external unverified library for cryptographic code. In general, this library could have bugs and may produce a

wrong result. This is not a problem per say because it will be caught during the certificate checking by any independent party, but it may create a atmosphere of distrust among voters.

Our formalization leaves some gaps which needs to be filled:

- A formally verified cryptographic library to fill the *correctness gap*.
- A formally verified checker to ease the auditing of election to fill the *scrutineers* gap

Developing a formally verified library to fill the correctness gap would have taken more time, specifically the commitment consistent shuffle, so we chose to formally verify the certificate checker to ease the auditing of election to increase the number of scrutineers gap.

In the next chapter, we will focus on all the details needed to develop formally verified certificate checker for the certificate we produced in this chapter. However, we did not formalize every cryptographic primitive needed to verify our certificate. Rather, we have developed a proof of concept formally verified certificate checker for International Association for Cryptologic Research 2018 election, a simpler scrutiny sheet than ours which does not involve any shuffle.

Chapter 6

Scrutiny Sheet : Software Independence

Somewhere inside of all of us is
the power to change the world.

Roald Dahl

6.1 Introduction

A major disadvantage of using cryptography to achieve privacy, using encryption to make the content of ballot private, and verifiability, using zero-knowledge proof for verification of claims, is that the verification process is quite cumbersome. As a consequence, the verification process (checking the scrutiny sheet) is only viable for cryptographers, a tiny fraction of general population, and results into a sharp decrease in number of scrutineers. While it is not very difficult to find cryptographers to verify the election, they are, off course, not the representative population in any democracy. In order to increase the number of scrutineers and subsequently confidence in electronic voting, we follow the route of providing a formally verified open-source reference certificate checker which anyone can inspect and run on the election data. The rationale behind formally verifying the certificate checker is *correctness* and open sourcing is to gain the public trust via careful examination. For example, consider a scenario where we do not provide the reference checker, then how likely would it be for community/voters to develop the verified checker? Moreover, assuming that we publish one unverified certificate checker, what would happen if it returns false on a valid certificate because of its own bug? Both situations, of course, would be a devastating situation, so not only should we provide a reference certificate checker, but it should be a formally verified one. Additionally, a formally verified reference certificate checker would open the gate for debate in case of someone's implementation for checking certificate diverges from the reference checker. In the case of a diverging situation, there are two possibilities, either the reference checker is verified using wrong assumptions, or the implementation itself is wrong. The first situation is certainly not very pleasant because it would deteriorate the public trust in the system, but nonetheless, it is always good to have openness in democracy to make it more strong.

In this chapter, we discuss the concepts required to develop a verified certificate checker for the certificate we generated in the last chapter. Moreover, we sketch pseudo code and pen-and-paper proof, in style of algebraic manipulation. The reason for doing this to make it accessible for everyone who intends to develop a formally verified certificate checker. In some cases, we have translated the pseudo code in Coq code to make the idea more precise.

We have already explained our certificate in section 5.5, but intuitively, checking our certificate amounts to proving that the homomorphic margin has been computed correctly and zero-knowledge proof for every claim is correct. In a nutshell, our claims were:

1. honest decryption zero-knowledge proof: every encrypted value is decrypted honestly

2. shuffle zero-knowledge proof: a ballot has been permuted by the same permutation whose commitment is published (commitment consistent shuffle [Wikström, 2009]).

3. final homomorphic tally is computed correctly

We sketch the encoding of Pedersen's commitment [Pedersen, 1992], one of the primitives of shuffle zero-knowledge proof, in Coq, but we leave the other details of shuffle zero-knowledge proof algorithm [Wikström, 2009]. Moreover, we encode the sigma protocol in Coq as a record and give various examples of concrete sigma protocol, including the honest decryption zero-knowledge proof. We also describe the computation of homomorphic tally from the encrypted ballots, without decryption any individual ballot. Finally, all these concepts are sufficient to develop the formally verified certificate checker for International Association for Cryptologic Research (IACR) election[1] because IACR uses a very simple method, compared to ours, to elect candidates, and our proof development can be accessed[2].

Chapter Outline: In section 6.2, we discuss the underlying algebraic structures needed for various cryptographic operations. Section 6.3 focuses on generalizing the Pedersen commitment scheme for a matrix. In the following section 6.4, we discuss the details of sigma protocol, and its formalization in Coq. In order to eschew the (monadic) probabilistic reasoning of sigma protocol, we use the standard trick of making the randomness explicit to make the reasoning easier without losing the meaning of sigma protocol. In section 6.4.1, we show how to make a concrete instance of sigma protocol by giving an example of discrete logarithm. In addition, we show the protocol needed for honest decryption (section 6.4.2). Section 6.5 sketches the homomorphically tally based on additive ElGamal scheme. Finally, we discuss the IACR scrutiny sheet in section 6.6, and we summarize the chapter in section 6.7

6.2 Algebraic Structures: Building Blocks

The basic building blocks of any cryptographic system are algebraic structures, specifically cyclic group (of prime order), field, and vector space. In general, we do not need vector space, and group and field are sufficient for most of the cryptographic purposes. However, vector space of a cyclic group of prime order over the field of integers (vector element) modulo the same field of integers of same prime order (scalar element) is nicer to work because the operation involving an element from group and an element from field can be abstracted over the scalar multiplication operator of vector space. For example, in elliptic curve cryptography, for a given curve E over a finite field, there are two main operations: i) point-addition (adding two points P and Q, given of the curve E), and ii) point-multiplication (multiplying a field element, a, with a point, P, on the curve E). Moreover, there is also a point at infinity (denoted by O) which acts as an identity element. We can easily implement these functions using the suitable data-structure from the Coq theorem prover and prove theorems about them. Nonetheless, during the process of proving these theorems a lot of details about the internal implementation

[1] https://vote.heliosvoting.org/helios/elections/60a714ea-ce6d-11e8-8248-76b4ab96574c/view

[2] https://github.com/mukeshtiwari/secure-e-voting-with-coq

of these functions would make it unnecessarily complicated[3], while if we abstract the point-addition and point-multiplication as vector-addition and scalar-multiplication, respectively, of a vector space, we can prove theorems about these functions, point-addition and point-multiplication, using just axioms of vector space. The proof process would be much smoother, but more importantly, the proof would just require the axioms of vector space and field. This was the major motivation behind abstracting the group and field over a vector space. Now we briefly explain the group, field, and vector space.

Definition 2 (Group) *A group is a set* G*, with a binary operator* $\cdot : G \to G \to G$*, identity element* e*, and inverse operator* $inv : G \to G$ *(denoted as* $^{-1}$*) such that the following laws hold:*

- *Associativity*: $\forall\ a\ b\ c\ \in G, a \cdot (b \cdot c) = (a \cdot b) \cdot c$
- *Closure*: $\forall\ a\ b\ \in G, a \cdot b \in G$
- *Inverse Element*: $\forall\ a\ \in G, a \cdot\ a^{-1} =\ a^{-1} \cdot a = e$
- *Identity*: $\forall\ a\ \in G, a \cdot e = e \cdot a = a$

If the group is commutative, i.e. $\forall\ a\ b\ \in G,\ a \cdot b = b \cdot a$, then we call it Abelian group. We can represent the Abelian group in Coq by using the record data type.

```
Record AbelGroup (G : Type)
    (dot : G -> G -> G)  (inv : G -> G) (e : G) :=
  {
    dot_associativity : forall x y z,
      dot x (dot y z) = dot (dot x y) z;
    dot_left : forall x, dot x e = x;
    dot_right : forall x, dot e x = x;
    left_inverse : forall x, dot (inv x) x = e;
    right_inverse : forall x, dot x (inv x) = e;
    commutative : forall x y, dot x y = dot y x
  }.
```

[3]it is more likely to be the case that some key lemma required would be missing from the library, and it will some effort to prove it

Our Coq encoding is slight different from the definition of the group we gave above, mainly that *G : Type* (read as G has the type Type). Because of the underlying foundation of Coq is type theory, every term in Coq has a type; hence we need to explicitly state the type of term G. In a nutshell, Type is used to represent set.

Definition 3 (Field) *A field is a set* $\mathbb{F}$*, with two binary operators* $+ : \mathbb{F} \to \mathbb{F} \to \mathbb{F}$*, and* $\cdot : \mathbb{F} \to \mathbb{F} \to \mathbb{F}$*, two identity elements* 0 *and* 1*, and two unary operator* $- : \mathbb{F} \to \mathbb{F}$*,* $1/ : \mathbb{F} \to \mathbb{F}$ *such that:*

- ($\mathbb{F}$, +, 0, -) forms an abelian group.
- ($\mathbb{F}$ - $\{0\}$, $\cdot$, 1, $1/$) forms an abelian group.
- $\cdot$ distributes over +.

Definition 4 (Vector Space) *A set* V *with two binary operations, vector addition* $+ : V \to V \to V$ *and scalar multiplication* $\cdot : \mathbb{F} \to V \to V$*, is a vector space over a field* $\mathbb{F}$ *if the following properties hold:*

- Closure under vector addition: (V, +) forms an abelian group.
- Scalar multiplication distributes with respect to vector addition: $\forall\ r \in \mathbb{F}$, $u\ v \in V$, $r \cdot (u + v) = r \cdot u + r \cdot v$.
- Scalar multiplication distributes with respect to field addition: $\forall\ a\ b \in \mathbb{F}$, $u \in V$, $(a +_{\mathbb{F}} b) \cdot u = a \cdot u + b \cdot v$, where $+_{\mathbb{F}} : \mathbb{F} \to \mathbb{F} \to \mathbb{F}$ is field addition.
- Scalar multiplication is associative with respect to field multiplication: $\forall\ a\ b \in \mathbb{F}$, $u \in V$, $(a \cdot_{\mathbb{F}} b) \cdot u = a \cdot (b \cdot u)$, where $\cdot_{\mathbb{F}} : \mathbb{F} \to \mathbb{F} \to \mathbb{F}$ is field multiplication.
- Identity: $\forall a \in V, 1_{\mathbb{F}} \cdot a = a$

6.3 Pedersen Commitment Scheme

Recall that in the last chapter to prove that if a ballot was valid or invalid, we generated a secret permutation π and published its commitment using

Pedersen commitment scheme [Pedersen, 1992]. Later, we use this to shuffle each row and column of the ballot by this (secret) permutation. In the shuffle algorithm [Wikström, 2009]), the data structure of permutation π is matrix (a permutation matrix to be precise). In this section, we discuss that how to generalized Pedersen commitment scheme for matrix data structure.

A Pedersen commitment for any two given group elements $g\ h \in G$, a message $m \in \mathbb{F}$ (the field of integers), and a random element $r \in \mathbb{F}$ (the field of integers) is $g^r \cdot h^m$ ($\cdot$ is the group operation). As we explained above that we can just work with group and field, but abstracting the group operation ($\cdot : G \to G \to G$) as a vector-addition and exponentiation operation ($\hat{}\ : G \to \mathbb{F} \to G$) as scalar-multiplication would make the proofs more tractable. Finally we can encode the Pedersen commitment in the Coq theorem prover (simplified for the presentation):

```
Definition ped_commitment {F G : Type} (H1 : Group G)
        (H2 : Field F) (H3 : Vector_Space F G)
        (^ : G -> F -> G) (dot : G -> G -> G)
        (g h : G) (r m : F) : G :=  dot (g ^ r) (h ^ m).
```

The Coq definition of Pedersen commitment assumes the existence of two types F and G, together with hypothesis that G is group, F is field, and both, F and G, forms a vector space (we are not giving any concrete implementation of group or field, but an abstract representation, assuming two abstract type G and F. During the code extraction, we instantiate all these types and operations with a concrete representation and discharge the proof obligation to make sure that the assumptions hold, very similar to the Schulze algorithm extraction where we instantiate the *Cand* type with concrete candidates and discharge all the proof obligation).

We can extend the Pedersen commitment to commit a vector instead of just a scalar. To commit a vector of n group elements $h_1, h_2 \dots h_n$, vector of n field elements $m_1, m_2 \dots m_n$, a group element g, and a random field element r, we compute $g^r \cdot \prod_{i=1}^{n} h_i^{m_i}$. This commitment is known as vector Pedersen commitment. The $\prod_{i=1}^{n} h_i^{m_i}$ can be computed in Coq as (this program is written using Equation library [Sozeau and Mangin, 2019]):

```
Equations prod_vh_commitment  {F G : Type} {n : nat}
        (H1 : Group G) (H2 : Field F)
        (H3 : Vector_Space F G)  (^ : G -> F -> G)
```

```
        (dot : G -> G -> G) (hi : Vector.t G (S n))
        (mi : Vector.t F (S n)) :  G :=
prod_vh_commitment (^) dot (vcons h vnil) (vcons m vnil) :=
        h ^ m;
prod_vh_commitment (^) dot (vcons h hs)  (vcons m ms)  :=
        dot (h ^ m) (prod_vh_commitment ^ dot hs ms).
```

Now we can compute the vector Pedersen commitment ($g^r \cdot \prod_{i=1}^{n} h_i^{m_i}$) as:

```
Definition ped_vec_commitment {F G : Type} {n : nat}
        (H1 : Group G) (H2 : Field F)
        (H3 : Vector_Space F G)  (^ : G -> F -> G)
        (g : G) (r : F) (hs : Vector.t G (n + 1))
        (ms : Vector.t F (n + 1)) :=
        dot (g ^ r)
        (prod_vh_commitment H1 H2 H3 ^ dot hs ms)
```

Finally, we can extend the idea of vector Pedersen commitment to commit a matrix of size $N \times N$, for some arbitrary natural number N. To do so, we can call the *ped_vec_commitment* on every column of the matrix. Consequently, we will get a vector of commitments of length N.

6.4 Sigma Protocol: Efficient Zero-Knowledge Proof

A sigma protocol is a two party protocol, a prover P and a verified V, where prover P tries to convince the verifier V that she holds a private input x for some public input w such that a binary relation R holds, i.e. $(x, w) \in R$. Sigma protocol, in general, is a three step protocol:

1. Initialisation: P generates a random challenge r, commits it, and sends the committed message to V
2. Challenge: V generates a random challenge e and sends it to V
3. Response: P sends a response z to V

Finally, upon receiving the response z and other public inputs, V either accepts the proof or rejects the proof, depending on if the public inputs are consistent with protocol or not. The verification step is modelled as a boolean function that takes all the public inputs and returns true or false. Now we define the sigma protocol in Coq by using record data type.

```
Record SigmaProtocol (Statement : Type) (* Statement x *)
       (Witness : Type) (* witness w *)
       (R : Statement -> Witness -> bool) (* decidable relation *)
       (RandCoin : Type) (* random coin *)
       (Commitment : Type) (* commitments *)
       (Challenge : Type) (* challenges *)
       (Response : Type) (* response *) :=
  MkSigma {
  (* initial commitment send by the Prover *)
  initial : RandCoin -> Commitment;
  (* Randomness send by the verifier.  *)
  challenge : Challenge;
  (* response generate by prover *)
  response : Statement -> Witness ->
     RandCoin -> Challenge ->
     Response;
  (* verify the response *)
  verify : Statement * Commitment * Challenge * Response
  -> bool;
  (* Simulator *)
  simulator : Statement -> Challenge -> Response ->
     Statement * Commitment * Challenge * Response;
  (* Extractor *)
  extractor : Challenge -> Response -> Challenge ->
     Response -> Witness;


  (* Completness *)
  Completness : forall (s : Statement) (w : Witness)
    (r : RandCoin) (e : Challenge),  R s w = true ->
    verify (s, initial r, e, response s w r e) = true;

 (* Special Soundness *)
  Special_Soundness : forall s c e1 e2 r1 r2,
    e1 <> e2 ->
    verify (s, c, e1, r1) = true ->
    verify (s, c, e2, r2) = true ->
```

```
    R s (extractor e1 r1 e2 r2) = true;

 (* Special honest verifier zero knowledge proof. Explicit
    randomness makes it nicer to work in theorem prover  *)
  Special_Honest_Verifier_ZKP  (s : Statement)
    (w : Witness) (e : Challenge):
    R s w = true -> forall (r : RandCoin),
    verify (s, initial r, e, response s w r e) = true <->
    forall (z : Response), verify (simulator s e z) = true;

 (* simulator correct *)
  Simulator_correct : forall (s : Statement)
   (e : Challenge) (r : Response),
   verify (simulator s e r) = true;
  }.
```

The record *SigmaProtocol* is indexed by:

- *Statement*, the public input known to P and V
- *Witness*, secret input known to P
- R such that $(x, w) \in R$, known to P and V
- *RandCoin*, the private random coin toss of P
- *Commitment*, commitment computed by P based on the random coin toss
- *Challenge*, the random challenge of V to P
- *Response*, the response of P send to V

The body of record SigmaProtocol contains functions *initia*, *challenge*, and *response* to reflect the three steps of sigma protocol with three auxiliary functions and *verify*, *simulator*, and *extractor*. The *verify*, a boolean function, checks if the data produced during the protocol is consistent or not, *simulator* function produces a transcript and used for proving the special honest verifier zero knowledge proof, and *extractor* function produces a witness and used in special soundness of sigma protocol.

We have four correctness properties about sigma protocol:

1. *Completeness* : if P and V follow the protocol, then verifier would accept the proof.

2. *Special_Soundness* : if P is able to convince V with two accepting transcript for the same commitment, then V can extract the witness.

3. *Special_Honest_Verifier_ZKP* : recall that special honest verifier zero knowledge proof amounts to a probabilistic polynomial time simulator S which would generate a proof transcript for some statement s with same probability distribution as if there were a real conversation between a prover P and a verifier V for the statement s and witness w such that $(s, w) \in R$. Informally, the real proof transcript depends on statement s, witness w, and challenge e, while the simulated proof transcript depends on statement s and challenge e. (simulator does have not access to witness w, so to generate a accepting proof just by using s and e, it uses a concept call rewinding.) In our definition of *Special_Honest_Verifier_ZKP*, we eschew the probabilistic reasoning by making randomness explicit, and it states that for any given fixed statement s, witness w, challenge e and assumption that $R\ s\ w$ holds, then for every random challenge r and a accepting real transcript, simulator can construct an accepting transcript from all random responses drawn from response space.

4. *Simulator_correct* : simulator is correct, i.e. any transcript created by simulator checks out.

Finally, we can use our construction, *SigmaProtocol*, as a building block for composing different kinds of sigma protocols, which we are not explaining here. For example, we can define AND composition, EQ composition, OR composition, etc.

6.4.1 Concrete Sigma Protocol: Discrete Logarithm

One of the most basic sigma protocol is proof of knowledge of discrete logarithm, i.e. given two elements g and h of a group G, prover convinces the verifier that she knows the discrete logarithm ($\log_g h$) in zero knowledge. In Camenisch-Stadler notation [Camenisch and Stadler, 1997] of zero knowledge proof, it is represented as: $ZKPoK\{w \mid h = g^w\}$. We can show that this is a sigma protocol inside Coq by encoding all the functions and proving all the axioms mentioned in our record type *SigmaProtocol*. For example, we can write the *initial* function as taking a input r and computing g^r, *challenge* as a function which simply returns a challenge e, and so forth:

initial g r := g^r

challenge := e

response h w r e := $r + e \cdot w$

verify g h a e z := $g^z = a \cdot h^e$

simulator g h s e z := $(g^z \cdot h^{-e}, e, z)$

extractor c_1 z_1 c_2 z_2 := $(z_1 - z_2) \cdot (c_2 - c_1)^{-1}$

Based on these definitions, we can easily discharge the three proofs, *Completeness*, *Special_Soundness*, *Special_Honest_Verifier_Zero_Knowledge*, and *Simulator_correct* axioms by simple algebraic manipulation.

6.4.2 Honest Decryption Zero Knowledge Proof

We have sigma protocol at our arsenal, we focus on honest decryption problem. How can we convince someone that for a given group (G, g, p, h) and private key x ($h := g^x$), the message m is the honest decryption of ElGamal ciphertext (c_1, c_2) (which is $(g^r, g^m \cdot h^r)$ for some randomness r with revealing our private key x? To solve this problem, we use a well known protocol for proving equality of the discrete logarithm [Cramer et al., 1997]. We first discuss the protocol, and later we will show that how we can adopt the protocol for our purpose.

Diffie Hellman Tuple: a tuple (g, h, u, v) is a Diffie Hellman tuple if there exists a w such that $u = g^w$ and $v = h^w$. The protocol to prove it is:

- P chooses a random r and sends $a = g^r$ and $b = h^r$.
- V sends a random e
- P sends $z = r + e \cdot w$
- V check $g^z = a \cdot u^e$ and $h^z = b \cdot v^e$

Now we come back to our original problem, i.e. proving that m is the honest decryption of (c_1, c_2). From these values, we construct a Diffie Hellman tuple by multiplying c_2 with g^{-m}, i.e. $(g, h, c_1, c_2 \cdot g^{-m})$. A simple algebraic simplification shows that this tuple can be written as $(g, h, g^r, g^m \cdot h^r \cdot g^{-m})$ for some random r. A further simplification leads to (g, h, g^r, h^r), and this tuple is clearly a Diffie Hellman tuple, where $u = g^r$ and $v = h^r$. We could not have been able to construct a Diffie Hellman tuple and proved the equality of discrete logarithm if we had claimed anything other than the origin value m. For example, suppose a cheating prover claims that m_1 (different from m) is the honest decryption of (c_1, c_2). Following the cheating prover claim, we construct the Diffie Hellman tuple $(g, h, g^r, g^m \cdot h^r \cdot g^{-m_1})$. Clearly, the tuple $(g, h, g^r, h^r \cdot g^{m-m_1})$ is not Diffie Hellman tuple; hence a cheating prover would not succeed.

6.5 Homomorphic Tally

Now that we have sorted out the correct decryption, the next challenge in our tally sheet is computing the final tally homomorphically. Since our encryption is additive ElGamal, and recall that our ballot is a matrix of ciphertexts:

$$\begin{pmatrix} (g^{r_{11}}, g^{m_{11}} * h^{r_{11}}) & (g^{r_{12}}, g^{m_{12}} * h^{r_{12}}) & \cdots & (g^{r_{1N}}, g^{m_{1N}} * h^{r_{1N}}) \\ (g^{r_{21}}, g^{m_{21}} * h^{r_{21}}) & (g^{r_{22}}, g^{m_{22}} * h^{r_{22}}) & \cdots & (g^{r_{2N}}, g^{m_{2N}} * h^{r_{2N}}) \\ \vdots & \vdots & \ddots & \vdots \\ (g^{r_{N1}}, g^{m_{N1}} * h^{r_{N1}}) & (g^{r_{N2}}, g^{m_{N2}} * h^{r_{N2}}) & \cdots & (g^{r_{NN}}, g^{m_{NN}} * h^{r_{NN}}) \end{pmatrix}$$

To compute the finally tally, all we have to do is to stack all the valid ballots (matrices) together and multiply the corresponding ciphertexts together to get the final tally matrix (point wise matrix multiplication). The final computed tally can be decrypted honestly by using the same principals described in the previous section. We can capture all these concepts in Coq based on the algebraic structures, group, field, vector space, and prove all the properties by simple algebraic manipulation. We can represent encryption, decryption and ciphertext multiplication for a given cyclic group (G, g, h, x) such that $h = g^x$:

elGamal_enc (g h : G) (r : F) :=$(g^r, g^m \cdot h^r)$

elGamal_dec (g h : G) $(c_1, c_2) := c_2 \cdot c_1^{-x}$

elGamal_mult $(c_1, c_2)(d_1, d_2) := (c_1 \cdot d_1, c_2 \cdot d_2)$

In fact, by simple algebraic manipulation, we can prove that decryption is left inverse of encryption.

$$
\begin{aligned}
\text{elGamal\_dec g h (elGamal\_enc g h r)} &= \text{elGamal\_dec g h } (g^r, g^m \cdot h^r)(unfolding) \\
&= g^m \cdot h^r \cdot (g^r)^{-x}(unfolding) \\
&= g^m \cdot (g^x)^r \cdot (g^r)^{-x}(substitution) \\
&= g^m \cdot g^{xr} \cdot g^{-rx}(algebraic - simplification) \\
&= g^m
\end{aligned}
$$

The final decrypted tally would be a matrix filled with values like $g^{m_1+m_2+\cdots}$, and we need to do a search to find the values of $m_1 + m_2 + \cdots$ from the final decrypted tally. A drawback of this method is that if the number of candidates and ballots are large, then calculating $m_1 + m_2 + \cdots$ from $g^{m_1+m_2+\cdots+m_n}$ is not very practical [Cramer et al., 1997].

6.6 IACR 2018 Election

We follow some of these techniques explained above to write a formal certificate checker for IACR 2018 directors election scrutiny sheet [4]. The 2018 IACR directors election considered seven candidates to fill three positions on the board of directors. The voting style is approval voting where all the eligible voters, IACR members, could vote for as many candidates as they like. After the counting, the top three members were elected to fill the positions.

The Helios voting system [Helios, 2016] was used for the election, and the system was configured with four authorities, who generated an ElGamal [ElGamal, 1985] public key such that all four authorities were required to decrypt efficiently. Every eligible voter received the credentials by email which they used to cast their ballot from their personal computer. During the cast process, each voter created seven ElGamal ciphertexts, encrypting either zero or one, for the seven participating candidates. Since the vote was exponent, the ElGamal cryptosystem became homomorphic additive. During this point, the voter was then offered the chance to audit her encrypted ballot to check that it indeed had the vote of her choice. If she chose to audit, she had to

[4] https://vote.heliosvoting.org/helios/elections/60a714ea-ce6d-11e8-8248-76b4ab96574c/view

discard this ballot and asked to cast a fresh ballot (this mechanism is called called Benaloh challenge, and the purpose of this challenge is to catch "cheating machines". Moreover, this method ensures the cast-as-intended because a cheating machine would not know when a voter would cast her ballot, so, in the most likely scenario, a voter would end up cast her true intentions). Once she had an unaudited ballot with which she was happy, she cast it. The Helios website maintained an append-only bulletin board on which the voter's encrypted ballot appeared. After the voting period was over, all the encrypted ballots corresponding to all candidates were multiplied together; so that there was a single ciphertext for each candidate, encoding the number of votes for that candidate. The authorities then decrypted these (seven) ciphertexts, announced the result and proved, using a sigma protocol, that the announced result was the correct decryption.

Now that we have already explained the workings of IACR 2018 directors election, we focus on three aspects of verifiability: cast-as-intended, collected-as-cast, and tallied-as-collected. The cast-as-intended has already been assured by Benaloh challenges, and collect-as-cast is ensured by every voter checking her ballot on the bulletin board. The more complicated step is the tallied-as-cast check. In order to verify the tallied-as-cast, a scrutineer has to check only the valid ballots (those which are encryption of either zero or one) has contributed to final tally, the final tally has been calculated correctly, and the final tally has been decrypted honestly.

The algorithm, in more detail, to ascertain the tallied-as-cast, there is a published list of encrypted ballots on the bulletin board and a published result. Moreover, to enable scrutiny, the election authority publishes, non-interactive, sigma protocol transcripts for correct encryption and decryption. Using these transcripts, the scrutineer can verify the election by checking the following three things. First, all the ballots included in final tally are indeed the encryption of zero or one, and any ballot containing any other value has been discarded. Second, the scrutineer reruns the (multiplication) computation and checks that the resulting ciphertexts matches the published one. Finally, she checks that the transcripts are valid for the decryption of these combined ciphertexts with respect to the announced result. These three checks suffice to ensure that the ballots were counted-as-collected.

IACR used Schnorr group [Schnorr, 1990] to avoid attacks various attacks on solving the discrete logarithm problem. A Schnorr group is a multiplicative Abelian subgroup of prime order q of the field of integers modulo a prime p, where $p = k * q + 1$ for some integer. In IACR election, the primes used were:

```
Definition P : Z := 16328632084933010002384055033805457329601614771
185955389739167309086214800406465799038583634953752941675645562182
4981207502649804923813755793676756487712938003103709647457670142436
3851844255382397348299526730404432677704766295748026939132278937838
4619428596446446984694306187644767462460965622580087564339212631775
8178959584090166763989756712661796378985576873170761772188432331506
9515788106125705301913307854592898356222139631316962247550981844266
1047018436264806901023966236718367204710755935899013750306107738002
3641379174265957374038711141877508043465647312506091968466381839039
8238788457826613650369749347468207 1.
```

Since theorem provers are known for proving mathematical statements, but not for being good at running computation inside their environment. Naturally, proving any mathematical statement, e.g. number theoretic proofs, which are computational intensive would not be a ideal situation for theorem provers. However, the recent advancement in theorem provers (specifically Coq) led us to prove primality of two large prime numbers inside the Coq. To begin with we utilise the CoqPrime library[5] to prove in Coq that the numbers used to define the Schnorr group are in fact prime.

```
Lemma P_prime : prime P.

Lemma Q_prime : prime Q.
```

Finally, we extract the OCaml code from Coq proof scripts and write a main file to glue the extracted code and parsing code. Upon execution, the code returns yes, which asserts that the results produced are correct.

6.7 Summary

In this chapter, we have sketched the ideas for developing a formally verified certificate checker for the certificate we produced in the last chapter. However, due to time constraint and complexity of shuffle primitive, we ended up verifying a simple certification, which did not involve any zero-knowledge proof of shuffle. Finally, in this chapter we closed the loop of decrease in

[5]https://github.com/thery/coqprime

number of scrutineers because any one can run the certificate checker. Moreover, we open sourced [6] the checker, so that it can be inspected by anyone (we would like to call it correctness by democratic process). One thing we would like to emphasize that cryptographic concepts are inherently very complex, so running a certificate checker certainly does not amounts to understanding the various bits of cryptography and formal method used to develop the certificate checker.

In the next chapter, we will discuss some of the properties of Schulze method which we have formalized in Coq.

[6] https://github.com/mukeshtiwari/secure-e-voting-with-coq

Chapter 7

Machine Checked Schulze Properties

> Stay Hungry. Stay Foolish.
>
> *Steve Jobs*

Since the beginning of democracy, social scientists are constantly looking for methods which would aggregate the individual choices to arrive at acceptable group decisions. In general, these acceptable group decisions are based on intuition of the society at that time, but not backed by mathematical theory. The first mathematical treatment to combine the individual choices (social mathematics) can be attributed to French philosopher and mathematician Marquis de Condorcet (Condorcet method, 1785) and his contemporary and co-national mathematician Jean-Charles de Borda (Borda count, 1770). However, the first formal system, foundational cornerstone of modern social choice theory, for collective preference was given by Kenneth Arrow. In 1950, Kenneth Arrow published a paper titled *A Difficulty in the Concept of Social Welfare* [Arrow, 1950b]. In this paper, Kenneth Arrow envisioned an axiomatic system having the following properties:

- Unrestricted domain
- Non-dictatorship
- Pareto efficiency
- Independence of Irrelevant Alternatives

Moreover, he showed no preferential voting method which can combine or aggregate the individual choices into a community wide ranking would have all the properties of his axiomatic system. This result is now known as *Arrow's impossibility theorem*.

In the light of impossibility theorem, Schulze method, a preferential voting method, can not have all the properties, and it fails on Independence of Irrelevant Alternatives (IIA) criterion. Despite the fact that Schulze method fails on IIA, it has plenty of other desirable properties established in the social choice theory. In this chapter, we will discuss some of the properties. Moreover, we will show that our implementation adheres to these properties.

7.1 Condorcet Winner

A *Condorcet winner* is a candidate who beats every other candidate in pairwise comparison (also known as head to head competition). Recall that in Schulze method, the pairwise comparison method was margin matrix, denoted as *marg*, which defined as:

> Given a set of ballots P and candidate set C, we construct graph G based on the margin matrix $marg : C \times C \to \mathbb{Z}$. Given two candidates $c, d \in C$, the *margin* of c over d is the number of voters that prefer c over d, minus the number of voters that prefer d over c. In symbols:
>
> $$marg(c, d) = \sharp\{b \in P \mid c >_b d\} - \sharp\{b \in P \mid d >_b c\}$$
>
> where $\sharp$ denotes cardinality and $>_b$ is the strict (preference) ordering given by the ballot $b \in P$.

Now we define the *Condorcet winner* in Coq as:

```
Definition condorcet_winner (c : cand)
  (marg : cand * cand -> Z) := forall d, marg (c, d) >= 0.
```

Informally, the definition, *condorcet_winner*, states that if a candidate c is *condorcet winner*, then she has been ranked higher against every other candidate. Having the definition of condorcet winner, our goal is to concluded that if there is a Condorcet winner, the Schulze method always elects it as a winner.

```
(* if candidate c is Condorcet winner then it's winner of election *)
 Lemma condorcet_winner_implies_winner (c : cand)
    (marg : cand * cand -> Z) :  condorcet_winner c marg ->
    c_wins marg c = true.
Proof.
      intros Hc.
      pose proof  condorcet_winner_genmarg.
      pose proof c_wins_true.
      apply H0. intros d.
      pose proof (H c d (length cand_all) marg Hc).
      auto.
Qed.
```

The proof of this theorem hinges on the two key facts:

1. If a canddiate beats everyone in pairwise comparison, then generalised margin between her and every other candidate would be greater than or equal to 0.

2. If a canddiate beats everyone in pairwise comparison, then generalised margin between every other candidate and her would be less than or equal to 0.

It is not very hard to see these two facts based on the definition of generalised margin. Intuitively, if a candidate c is the Condorcet winner, then the strongest path between her and every other candidate, say d, would be either a direct path, $marg(c, d)$, or a more stronger path, $M(c, d)$, via some other intermediate candidates.

A directed *path* in the graph, G, from candidate c to candidate d is a sequence $p \equiv c_0, \ldots, c_{n+1}$ of candidates with $c_0 = c$ and $c_{n+1} = d$ ($n \geq 0$), and the *strength*, st, of path, p, is the minimum margin of adjacent nodes, i.e.

$$st(c_0, \ldots, c_{n+1}) = \min\{marg(c_i, c_{i+1}) \mid 0 \leq i \leq n\}.$$

For candidates c and d, let $M(c,d)$ denote the maximum strength, or generalised margin of a path from c to d i.e.

$$M(c,d) = \max\{st(p) : p \text{ is path from } c \text{ to } d \text{ in G}\}$$

We capture these two facts in Coq:

```
 Lemma gen_marg_gt0 :
   forall c d  marg,
   condorcet_winner c marg ->
   M (c, d) >= 0.
Proof.
   unfold condercet_winner.
   intros c d n marg Hc.
   rewrite M_M_new_equal.
   revert d; revert n.
   induction n; cbn; try auto.
   intros d. pose proof (IHn d).
   lia.
Qed.

Lemma gen_marg_lt0 :
   forall c d marg ,
   condorcet_winner c marg ->
   M (d, c) <= 0.
Proof.
  unfold condercet_winner.
  intros c d n marg Hc.
  rewrite M_M_new_equal.
  revert d; revert n.
  induction n.
  + cbn. intros d. pose proof (marg_neq c d marg).
      pose proof (Hc d). lia.
   + cbn. intros d.
       apply Z.max_lub_iff.
       split.
       pose proof (IHn d). lia.
       apply upperbound_of_nonempty_list; try auto.
       intros x Hx. pose proof (IHn x).
       lia.
Qed.
```

Using these two key facts, we conclude that for any Condorcet winner candidate c, the generalised margin between her and every other opponent is greater than or equal to generalised margin between every other candidate and her. Formally, in Coq we prove the following theorem, *condorcet_winner_genmarg*, on which the proof of *condorcet_winner_implies_winner* hinges.

```
Lemma condorcet_winner_genmarg :
     forall c d marg,
     condorcet_winner c marg ->
     M  (d, c) <= M (c, d).
Proof.
  intros c d n marg Hc.
  pose proof (gen_marg_gt0 c d n marg Hc).
  pose proof (gen_marg_lt0 c d n marg Hc).
  lia.
Qed.
```

7.2 Reversal Symmetry

The *Reversal symmetry* is a voting method criterion which states that if the voting method has produced a unique winner, say c, based on the cast ballots, then c should not be elected if the individual choices were reversed. In context of Schulze method, we first need to define the unique winner, and ballot reversal.

```
Definition unique_winner
(marg : cand * cand -> Z) (c : cand) :=
 c_wins marg  c = true /\
 (forall d, d <> c -> c_wins marg d = false).
```

Informally, our definition of *unique_winner* states that the candidate c is a unique winner if it wins the election with respect to computed margin matrix, *marg*, and every candidate other than c loses the election.

We capture the ballot reversal in terms of margin matrix. For any given ballot set, if the computed margin between two candidates c and d is:

$$marg(c,d) = \sharp\{b \in P \mid c >_b d\} - \sharp\{b \in P \mid d >_b c\}$$

If we reverse each ballot from the ballot set, then the new margin matrix, denoted as *rev_marg*, would be:

$$rev\_marg(c,d) = -1 * marg(c,d)$$

This fact can also be demonstrated using a single ballot ABC. The interpretation is A is strictly preferred over B, and B is preferred over C (but we do not need strict preferences to have this property). The margin matrix constructed from this ballot is:

$$\begin{array}{c c} & \begin{array}{ccc} A & B & C \end{array} \\ \begin{array}{c} A \\ B \\ C \end{array} & \begin{pmatrix} 0 & 1 & 1 \\ -1 & 0 & 1 \\ -1 & -1 & 0 \end{pmatrix} \end{array}$$

After reversing the original ballot, we get CBA and the margin matrix is:

$$\begin{array}{c c} & \begin{array}{ccc} A & B & C \end{array} \\ \begin{array}{c} A \\ B \\ C \end{array} & \begin{pmatrix} 0 & -1 & -1 \\ 1 & 0 & -1 \\ 1 & 1 & 0 \end{pmatrix} \end{array}$$

We capture this notion formally in Coq as:

```
Definition rev_marg
   (marg : cand -> cand -> Z) (c d : cand) :=
   -marg c d.
```

Based on the our definition of *rev_marg*, we can formally state the reversal symmetry as:

```
Lemma winner_reversed :
      forall marg c, unique_winner marg c ->
      c_wins (rev_marg marg) c = false.
```

The lemma, *winner_reversed*, expresses that if a candidate c is a unique winner with respect to *marg* (computed from some ballot set P), then she is not a winner with respect to *rev_marg* (computed from reversing all the entries in the ballot set P).

The proof for this lemma is fairly intuitive, but it takes some efforts to prove it in Coq. In this lemma, we assume the existence of unique winner, say c with respect to *marg*, which means that the generalised margin between her and every other candidate would be greater than between the every other candidate and her, i.e. $\forall d,\ M(c,d) > M(d,c)$. If we compute the generalize margin with respect to *rev_marg*, then for the candidate c it would be the case that: $\forall d,\ M\_rev(c,d) < M\_rev(d,c)$. One key observation is that the graph we get after computing the generalised margin with respect to *rev_marg* is simply a mirror image, every path is reversed, of the graph we get after computing the generalize margin with respect to *marg*. In terms of Coq, it is:

```
Lemma path_with_rev_marg :
  forall k marg c d,
  Path marg k c d <->  Path (rev_marg marg) k d c.
Proof.
  intros k marg c d.
  split. intro H.
  destruct (path_iterated_marg marg k c d H) as [n Hn].
  destruct (proj1 (iterated_marg_char marg n c d k) Hn)
    as [l [H1 H2]].
  rewrite  str_and_rev_str in H2.

  assert (length (rev l) <= n)%nat.
  rewrite rev_length. auto.
  pose proof (path_len_iterated_marg
     (rev_marg marg) n d c k (rev l) H0 H2).
  pose proof (iterated_marg_path
    (rev_marg marg) n k d c H3). auto.

  intros H.
  destruct (path_iterated_marg (rev_marg marg) k d c H)
    as [n Hn].
  destruct (proj1 (iterated_marg_char
    (rev_marg marg) n d c k) Hn) as [l [H1 H2]].
  apply iterated_marg_path with (n := length l).
  apply path_len_iterated_marg with (l := rev l).
  rewrite rev_length. lia.
  rewrite str_and_rev_str.
```

```
  rewrite rev_involutive. auto.
 Qed.
```

Using the lemma *path_with_rev_marg*, we conclude that if the strength of a path going from *c* to *d* with respect to *marg* is greater than or equal to *k* (by definition of *Path* inductive type), then the strength of a reverse path from *d* to *c* would also be greater than or equal to *k* with respect *rev_marg*. Using all these facts, and some auxiliary lemma the proof of reversal symmetry is merely rewriting the facts [1].

7.3 Summary

Although, we have just proved two properties of Schulze method, and so far, this chapter is far from being complete. The rationale behind this chapter was to put forward the idea of not only implementing the voting method and proving its correctness, but also proving that the implementation follows the property of voting method. In the next chapter, I will conclude this thesis and some possible direction for future work.

[1] At the time of writing this thesis, proof of reversal symmetry hinges on a auxiliary lemma which is fairly intuitive, but demands a lot of Coq machinery (a typical situation in theorem proving). We are in the process of proving it

Chapter 8

Conclusion and Future Work

> Education is the most powerful weapon which you can use to change the world.
>
> *Nelson Mandela*

This chapter summarizes the key outcomes of this dissertation, followed by possible future work.

8.1 Conclusion

Recall that the journey started with the purpose to make the electronic voting process transparent and trustworthy. The premise was:

> Given the potential advantages of electronic voting, we need to address correctness, privacy, and verifiability concerns for its widespread adoption.

The current state of art software program used by many governments is mostly black-box which takes a pile of ballots and produces result. To improve the current situation, we focussed on answering the four main concerns: *(i) correctness*, *(ii) privacy*, *(iii) coercion resistance*, and *(iv) verifiability (tallied-as-cast)* by using *Schulze method* as an example.

8.1.1 Correctness

With the intentions to solve the correctness in electronic voting, we approached from mathematical logic route. Rather than implementing the Schulze method, we gave a logical specification of the Schulze winning condition and losing condition with respect to an already computed margin function. These specifications were simple enough that one can inspect them and make sure that the intent is captured correctly. Moreover, we proved the correctness properties about our specification. Also, we put forward the idea of formalizing the properties of voting protocol, in our case Schulze method, in the framework developed by Kenneth Arrow in theorem prover itself. For example, in the last chapter, we proved the *Condorcet* winner property of Schulze method. Given that ballot counting is one of the most crucial phase of any election, the correctness of counting software should be explored from all the possible directions.

8.1.2 Verifiability

We answered the verifiability issue by producing a independently checkable scrutiny sheet. In our case, it contained the step by step computation of margin, winners and loser with the proof why they win or lose the election. This scrutiny sheet can be used any independent third party to verify the outcome of election. In case of plaintext ballot, achieving verifiability was trivial, but the encrypted ballot case was very complex. The reason for complexity was the inherent nature of cryptography, so to keep the encrypted ballot election verifiable we augmented the scrutiny sheet with zero-knowledge proofs. Finally, we developed a formally verified certificate checker to ease the auditing of election (although, we ended up developing checker for a different election).

8.1.3 Privacy and Coercion Resistance

Our approach to privacy and coercion problem was homomorphic encryption. To keep the content of ballot private, we did not decrypt any individual ballot and computed the final tally homomorphically by multiplying the ciphertexts. In the final step, we decrypted the fully computed tally, which in turn did not reveal any individual ballots. Since there was no decryption of any individual ballot, there was no way a voter could have convinced any one about her choices.

8.2 Future Work

8.2.1 Formalizing Cryptographic Entities

During the formalization, we assumed the cryptographic primitives for various construction, e.g. encryption primitive, decryption primitive, zero-knowledge-proof primitive, etc. Moreover, we assumed axioms about their correctness property. Formalizing all these primitives and proving the axioms we assumed would further close the trust gap.

8.2.2 Formalizing Properties of Schulze Method

We have formalized just two properties, *Condorcet winner*, and *Reversal symmetry*, but taking it further and proving all the properties would put more trust in the implementation. Moreover, it would be a good stress testing for the specification/implementation and see how far it can go.

8.2.3 Formally Verified Checker

Another interesting avenue would be to explore the formally verified certificate checker. In our formally verified certificate checker, we extracted an OCaml code for certificate checking. However, we wrote a substantial amount of unverified OCaml code for parsing the scrutiny sheet. To alleviate this kind of concerns, it would be worth exploring a verified parser, and, probably, evaluating the whole certificate inside Coq environment. There has already been verified parsers written in Coq [Jourdan et al., 2012], and given that we proved the two large primes inside a Coq, it does not seem a far fetched concept.

8.2.4 Risk Limiting Audit for Preferential Voting Scheme

One challenging potential opportunity would be developing a risk limiting audit system for the Schulze method. In nutshell, risk limiting audit is a method to audit the election by randomly sampling the ballots. Risk limiting is very well understood in the first-past-the-post voting system, and some of the work has been done in the context IRV elections [Blom et al., 2018], but so far none, to the best of my knowledge, for Schulze method.

8.2.5 Formalizing Code Extraction

This is orthogonal, but very important from the perspective of electronic voting. One of the biggest concern with code extraction of Coq in OCaml/Haskell is that all the security properties proved inside Coq are no longer valid in OCaml/Haskell. Taking the mission critical importance of electronic voting into account, we need a mechanism to translate the proofs all the way up to assembly level. CertiCoq [Anand et al.] seems promising, but it is not very mature yet. The other possibility is using the CakeML [Kumar et al., 2014] to develop the electronic voting schemes.

Bibliography

e-governance. https://e-estonia.com/solutions/e-governance/i-voting/. Accessed on October 17, 2019. (cited on page 2)

Electronic Voting. https://www.e-voting.cc/en/it-elections/world-map. Accessed on October 17, 2019. (cited on page 13)

German Constitution. https://www.bundesverfassungsgericht.de/SharedDocs/Entscheidungen/EN/2009/03/cs20090303_2bvc000307en.html. Accessed on October 18, 2019. (cited on page 15)

New South Wales Election. https://www.abc.net.au/news/2015-02-04/computer-voting-may-feature-in-march-nsw-election/6068290. Accessed on October 18, 2019. (cited on page 17)

Voting Computer Tempest Attack (Information Leaking). https://www.youtube.com/watch?v=B05wPomCjEY. Accessed on October 17, 2019. (cited on page 16)

Western Australia Senate Election. https://www.theguardian.com/world/2014/feb/28/western-australia-senate-election-re-run-to-be-held-on-5-april. Accessed on October 18, 2019. (cited on page 13)

Abadi, M. and Fournet, C., 2001. Mobile values, new names, and secure communication. *SIGPLAN Not.*, 36, 3 (Jan. 2001), 104–115. doi:10.1145/373243.360213. https://doi.org/10.1145/373243.360213. (cited on page 8)

Adrian, D.; Bhargavan, K.; Durumeric, Z.; Gaudry, P.; Green, M.; Halderman, J. A.; Heninger, N.; Springall, D.; Thomé, E.; Valenta, L.; VanderSloot, B.; Wustrow, E.; Zanella-Béguelin, S.; and Zimmermann, P., 2015. Imperfect forward secrecy: How diffie-hellman fails in practice. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15 (Denver, Colorado, USA, 2015), 5–17. ACM, New York, NY, USA. doi:10.1145/2810103.2813707. http://doi.acm.org/10.1145/2810103.2813707. (cited on pages 43 and 45)

Aguillon, J. Ocaml ↔ Java Interface. https://github.com/Julow/ocaml-java. Accessed on April 29, 2019. (cited on page 104)

Alkassar, E.; Böhme, S.; Mehlhorn, K.; and Rizkallah, C., 2014. A framework for the verification of certifying computations. *Journal of Automated Reasoning*, 52, 3 (Mar 2014), 241–273. doi:10.1007/s10817-013-9289-2. https://doi.org/10.1007/s10817-013-9289-2. (cited on pages 23 and 24)

Anand, A.; Appel, A.; Morrisett, G.; Paraskevopoulou, Z.; Pollack, R.; Belanger, O. S.; Sozeau, M.; and Weaver, M. Certicoq: A verified compiler for Coq. (cited on page 138)

Anand, A. and Rahli, V., 2014. Towards a formally verified proof assistant. In *Interactive Theorem Proving*, 27–44. Springer International Publishing, Cham. (cited on page 42)

Appel, A. W.; Michael, N.; Stump, A.; and Virga, R., 2003. A trustworthy proof checker. *Journal of Automated Reasoning*, 31, 3 (Nov 2003), 231–260. doi:10.1023/B:JARS.0000021013.61329.58. https://doi.org/10.1023/B:JARS.0000021013.61329.58. (cited on page 42)

Aranha, D. F.; Barbosa, P. Y.; Cardoso, T. N.; Araújo, C. L.; and Matias, P., 2019. The return of software vulnerabilities in the Brazilian voting machine. *Computers Security*, 86 (2019), 335 – 349. doi:https://doi.org/10.1016/j.cose.2019.06.009. http://www.sciencedirect.com/science/article/pii/S0167404819301191. (cited on page 2)

Arkoudas, K. and Rinard, M. C., 2005. Deductive runtime certification. *Electr. Notes Theor. Comput. Sci.*, 113 (2005), 45–63. (cited on page 22)

Arrow, K. J., 1950a. A difficulty in the concept of social welfare. *Journal of Political Economy*, 58, 4 (1950), 328–346. (cited on page 4)

Arrow, K. J., 1950b. A difficulty in the concept of social welfare. *Journal of political economy*, 58, 4 (1950), 328–346. (cited on page 127)

Australian Electoral Commission, 2013. Letter to Mr Michael Cordover, LSS4883 Outcome of Internal Review of the Decision to Refuse your FOI Request no. LS4849. avaliable via http://www.aec.gov.au/information-access/foi/2014/files/ls4912-1.pdf, retrieved October 23, 2019. (cited on pages 2 and 18)

Avgerou, C.; Masiero, S.; and Poulymenakou, A., 2019. Trusting e-voting amid experiences of electoral malpractice: The case of Indian elections. *Journal of Information Technology*, 34, 3 (2019), 263–289. doi:10.1177/0268396218816199. (cited on page 24)

Backes, M.; Hritcu, C.; and Maffei, M., 2008. Automated verification of remote electronic voting protocols in the applied pi-calculus. In *Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium*, CSF '08, 195–209. IEEE Computer Society, Washington, DC, USA. doi:10.1109/CSF.2008.26. https://doi.org/10.1109/CSF.2008.26. (cited on page 8)

Barendregt, H. P., 1992. Handbook of logic in computer science (vol. 2). chap. Lambda Calculi with Types, 117–309. Oxford University Press, Inc., New York, NY, USA. ISBN 0-19-853761-1. http://dl.acm.org/citation.cfm?id=162552.162561. (cited on pages 28 and 31)

Barras, B., 1996. Coq en coq. (1996). (cited on page 42)

Bayer, S. and Groth, J., 2012. Efficient zero-knowledge argument for correctness of a shuffle. In *Proc. EUROCRYPT 2012*, vol. 7237 of *Lecture Notes in Computer Science*, 263–280. Springer. (cited on pages 86 and 92)

Beckert, B.; Goré, R.; Schürmann, C.; Bormer, T.; and Wang, J., 2014. Verifying voting schemes. *Journal of Information Security and Applications*, 19, 2 (2014), 115 – 129. doi:https://doi.org/10.1016/j.jisa.2014.04.005. http://www.sciencedirect.com/science/article/pii/S2214212614000246. (cited on page 18)

Bellare, M. and Goldreich, O., 1993. On defining proofs of knowledge. In *Advances in Cryptology — CRYPTO' 92*, 390–420. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 50)

Ben-Or, M.; Goldreich, O.; Goldwasser, S.; Håstad, J.; Kilian, J.; Micali, S.; and Rogaway, P., 1988. Everything provable is provable in zero-knowledge. In *CRYPTO*, vol. 403 of *Lecture Notes in Computer Science*, 37–56. Springer. (cited on page 86)

Benaloh, J., 1994. Dense probabilistic encryption. In *Proceedings of the workshop on selected areas of cryptography*, 120–128. (cited on page 47)

Benaloh, J.; Moran, T.; Naish, L.; Ramchen, K.; and Teague, V., 2009. Shuffle-sum: coercion-resistant verifiable tallying for STV voting. *IEEE Trans. Information Forensics and Security*, 4, 4 (2009), 685–698. (cited on page 9)

Benaloh, J. and Tuinstra, D., 1994. Receipt-free secret-ballot elections (extended abstract). In *Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing*, STOC '94 (Montreal, Quebec, Canada, 1994), 544–553. ACM, New York, NY, USA. doi:10.1145/195058.195407. http://doi.acm.org/10.1145/195058.195407. (cited on pages 2 and 75)

Berardi, S., 1988. Towards a mathematical analysis of the coquand-huet calculus of constructions and the other systems in barendregt's cube. *Technica1 report, Carnegie-Me11on University (USA) and Universita di Torino (Ita1y)*, (1988). (cited on pages 28 and 31)

Bernhard, M.; Benaloh, J.; Halderman, J. A.; Rivest, R. L.; Ryan, P. Y. A.; Stark, P. B.; Teague, V.; Vora, P. L.; and Wallach, D. S., 2017. Public evidence from secret ballots. In *Proc. E-Vote-ID 2017*, vol. 10615 of *Lecture Notes in Computer Science*, 84–109. Springer. (cited on pages 2, 21, 75, and 86)

Bertot, Y.; Castéran, P.; Huet, G.; and Paulin-Mohring, C., 2004. *Interactive theorem proving and program development : Coq'Art : the calculus of inductive constructions*. Texts in theoretical computer science. Springer. (cited on pages 4, 27, and 54)

Blom, M.; Stuckey, P. J.; and Teague, V. J., 2018. Ballot-polling risk limiting audits for irv elections. In *Electronic Voting*, 17–34. Springer International Publishing, Cham. (cited on page 138)

Bonham, K. Most Tasmanian Senate Votes Were Unique. http://kevinbonham.blogspot.com/2019/06/most-tasmanian-senate-votes-were-unique.html. Accessed on 13 February, 2021. (cited on page 25)

Bruni, A.; Drewsen, E.; and Schürmann, C., 2017. Towards a mechanized proof of selene receipt-freeness and vote-privacy. In *Electronic Voting*, 110–126. Springer International Publishing, Cham. (cited on page 8)

Camenisch, J. and Stadler, M., 1997. Proof systems for general statements about discrete logarithms. *Technical report/Dept. of Computer Science, ETH Zürich*, 260 (1997). (cited on page 120)

Carré, B. A., 1971. An algebra for network routing problems. *IMA Journal of Applied Mathematics*, 7, 3 (1971), 273. (cited on page 68)

Chaum, D., 2004. Secret-ballot receipts: True voter-verifiable elections. *IEEE Security & Privacy*, 2, 1 (2004), 38–47. (cited on page 75)

Chaum, D. and Pedersen, T. P., 1992. Wallet databases with observers. In *CRYPTO*, vol. 740 of *Lecture Notes in Computer Science*, 89–105. Springer. (cited on page 90)

Chen, H.; Ziegler, D.; Chajed, T.; Chlipala, A.; Kaashoek, M. F.; and Zeldovich, N., 2015. Using crash hoare logic for certifying the fscq file system. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15 (Monterey, California, 2015), 18–37. ACM, New York, NY, USA. doi:10.1145/2815400.2815402. http://doi.acm.org/10.1145/2815400.2815402. (cited on page 19)

Chlipala, A., 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press. ISBN 0262026651, 9780262026659. (cited on page 54)

Cochran, D. and Kiniry, J., 2010. Votail: A formally specified and verified ballot counting system for Irish PR-STV elections. In *Pre-proceedings of the 1st International Conference on Formal Verification of Object-Oriented Software (FoVeOOS)*. (cited on page 8)

Cochran, D. and Kiniry, J. R., 2013. Formal model-based validation for tally systems. In *Proc. Vote-ID 2013*, vol. 7985, 41–60. Springer. (cited on page 9)

Cohen, E.; Dahlweid, M.; Hillebrand, M.; Leinenbach, D.; Moskal, M.; Santen, T.; Schulte, W.; and Tobies, S., 2009. Vcc: A practical system for verifying concurrent c. In *Theorem Proving in Higher Order Logics*, 23–42. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 24)

Constable, R. L.; Allen, S. F.; Bromley, H. M.; Cleaveland, W. R.; Cremer, J. F.; Harper, R. W.; Howe, D. J.; Knoblock, T. B.; Mendler, N. P.; Panangaden, P.; Sasaki, J. T.; and Smith, S. F., 1986. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. ISBN 0-13-451832-2. (cited on page 27)

Coquand, T. and Huet, G., 1988. The calculus of constructions. *Inf. Comput.*, 76, 2-3 (Feb. 1988), 95–120. doi:10.1016/0890-5401(88)90005-3. http://dx.doi.org/10.1016/0890-5401(88)90005-3. (cited on pages 5 and 28)

Coquand, T. and Paulin, C., 1988. Inductively defined types. In *International Conference on Computer Logic*, 50–66. Springer. (cited on page 5)

Cortier, V.; Dreier, J.; Gaudry, P.; and Turuani, M., 2019. A simple alternative to Benaloh challenge for the cast-as-intended property in Helios/Belenios. https://hal.inria.fr/hal-02346420. Working paper or preprint. (cited on page 3)

Cortier, V. and Smyth, B., 2011. Attacking and fixing helios: An analysis of ballot secrecy. In *2011 IEEE 24th Computer Security Foundations Symposium*, 297–311. doi:10.1109/CSF.2011.27. (cited on page 8)

Cortier, V. and Wiedling, C., 2012. A formal analysis of the norwegian e-voting protocol. In *Principles of Security and Trust*, 109–128. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 8)

Cramer, R.; Damgård, I.; and Schoenmakers, B., 1994. Proofs of partial knowledge and simplified design of witness hiding protocols. In *Advances in Cryptology — CRYPTO '94*, 174–187. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on pages 50, 51, and 86)

Cramer, R.; Gennaro, R.; and Schoenmakers, B., 1997. A secure and optimally efficient multi-authority election scheme. In *Advances in Cryptology — EUROCRYPT '97*, 103–118. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on pages 48, 121, and 123)

de Bruijn, N. G., 1983. *AUTOMATH, a Language for Mathematics*, 159–200. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-642-81955-1. doi:10.1007/978-3-642-81955-1_11. https://doi.org/10.1007/978-3-642-81955-1_11. (cited on page 27)

de Moura, L.; Kong, S.; Avigad, J.; van Doorn, F.; and von Raumer, J., 2015. The lean theorem prover (system description). In *Automated Deduction - CADE-25*, 378–388. Springer International Publishing, Cham. (cited on page 27)

Delahaye, D., 2000. A tactic language for the system Coq. In *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning*, LPAR'00 (Reunion Island, France, 2000), 85–95. Springer-Verlag, Berlin, Heidelberg. (cited on page 5)

Delaune, S.; Kremer, S.; and Ryan, M., 2010a. Verifying privacy-type properties of electronic voting protocols: A taster. In *Towards Trustworthy Elections, New Directions in Electronic Voting*, vol. 6000 of *Lecture Notes in Computer Science*, 289–309. Springer. (cited on pages 2 and 75)

Delaune, S.; Kremer, S.; and Ryan, M., 2010b. *Verifying Privacy-Type Properties of Electronic Voting Protocols: A Taster*, 289–309. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-642-12980-3. doi:10.1007/978-3-642-12980-3_18. https://doi.org/10.1007/978-3-642-12980-3_18. (cited on page 8)

DeYoung, H. and Schürmann, C., 2012. Linear logical voting protocols. In *Proc. VoteID 2011*, vol. 7187 of *Lecture Notes in Computer Science*, 53–70. Springer. (cited on pages 8 and 9)

Diffie, W. and Hellman, M., 2006. New directions in cryptography. *IEEE Trans. Inf. Theor.*, 22, 6 (Sep. 2006), 644–654. doi:10.1109/TIT.1976.1055638. http://dx.doi.org/10.1109/TIT.1976.1055638. (cited on pages 42 and 43)

Dijkstra, E. W., 1972. The humble programmer. *Commun. ACM*, 15, 10 (Oct. 1972), 859–866. doi:10.1145/355604.361591. http://doi.acm.org/10.1145/355604.361591. (cited on page 18)

Dolev, D. and Yao, A., 1983. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29, 2 (March 1983), 198–208. doi:10.1109/TIT.1983.1056650. (cited on page 8)

ElGamal, T., 1985. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31, 4 (1985), 469–472. (cited on pages 43, 45, 95, and 123)

Erbsen, A.; Philipoom, J.; Gross, J.; Sloan, R.; and Chlipala, A., 2019. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, 1202–1219. doi:10.1109/SP.2019.00005. https://doi.org/10.1109/SP.2019.00005. (cited on page 19)

Feldman, A. J.; Halderman, J. A.; and Felten, E. W., 2007. Security analysis of the diebold accuvote-ts voting machine. In *Proceedings of the USENIX Workshop on Accurate Electronic Voting Technology*, EVT'07 (Boston, MA, 2007), 2–2. USENIX Association, Berkeley, CA, USA. http://dl.acm.org/citation.cfm?id=1323111.1323113. (cited on page 2)

Fiat, A. and Shamir, A., 1987. How to prove yourself: Practical solutions to identification and signature problems. In *Proceedings on Advances in Cryptology—CRYPTO '86* (Santa Barbara, California, USA, 1987), 186–194. Springer-Verlag, Berlin, Heidelberg. (cited on page 51)

Firsov, D. and Uustalu, T., 2015a. Dependently typed programming with finite sets. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming*, WGP 2015 (Vancouver, BC, Canada, 2015), 33–44. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2808098.2808102. https://doi.org/10.1145/2808098.2808102. (cited on page 34)

Firsov, D. and Uustalu, T., 2015b. Dependently typed programming with finite sets. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, WGP@ICFP 2015, Vancouver, BC, Canada, August 30, 2015*, 33–44. doi:10.1145/2808098.2808102. https://doi.org/10.1145/2808098.2808102. (cited on page 63)

Fujioka, A.; Okamoto, T.; and Ohta, K., 1993. A practical secret voting scheme for large scale elections. In *Advances in Cryptology — AUSCRYPT '92*, 244–251. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 8)

GALINDO, D.; GUASCH, S.; AND PUIGGALÍ, J., 2015. 2015 neuchâtel's cast-as-intended verification mechanism. In *E-Voting and Identity*, 3–18. Springer International Publishing, Cham. (cited on page 3)

GENTRY, C., 2009. *A Fully Homomorphic Encryption Scheme*. Ph.D. thesis, Stanford, CA, USA. AAI3382729. (cited on page 46)

GEUVERS, H., 2001. Induction is not derivable in second order dependent type theory. In *Typed Lambda Calculi and Applications*, 166–181. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 32)

GEUVERS, H.; WIEDIJK, F.; AND ZWANENBURG, J., 2002. A constructive proof of the fundamental theorem of algebra without using the rationals. In *Types for Proofs and Programs*, 96–111. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 19)

GHALE, M. K.; GORÉ, R.; AND PATTINSON, D., 2017. A formally verified single transferable vote scheme with fractional values. In *Proc. E-Vote-ID 2017*, LNCS. Springer. This volume. (cited on pages 8 and 9)

GHALE, M. K.; PATTINSON, D.; KUMAR, R.; AND NORRISH, M., 2018. Verified certificate checking for counting votes. In *Verified Software. Theories, Tools, and Experiments*, 69–87. Springer International Publishing, Cham. (cited on page 9)

GIMÉNEZ, E., 1995. Codifying guarded definitions with recursive schemes. In *Types for Proofs and Programs*, 39–59. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 34)

GIRARD, J.-Y., 1987. Linear logic. *Theoretical Computer Science*, 50, 1 (1987), 1 – 101. doi:https://doi.org/10.1016/0304-3975(87)90045-4. http://www.sciencedirect.com/science/article/pii/0304397587900454. (cited on page 9)

GOLDREICH, O.; MICALI, S.; AND WIGDERSON, A., 1991. Proofs that yield nothing but their validity for all languages in NP have zero-knowledge proof systems. *J. ACM*, 38, 3 (1991), 691–729. (cited on page 86)

GOLDWASSER, S.; MICALI, S.; AND RACKOFF, C., 1985. The knowledge complexity of interactive proof-systems (extended abstract). In *STOC*, 291–304. ACM. (cited on pages 49 and 86)

GONTHIER, G., 2008. The four colour theorem: Engineering of a formal proof. In *Computer Mathematics*, 333–333. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 19)

Gu, L.; Vaynberg, A.; Ford, B.; Shao, Z.; and Costanzo, D., 2011. Certikos: a certified kernel for secure cloud computing. In *APSys '11 Asia Pacific Workshop on Systems, Shanghai, China, July 11-12, 2011*, 3. doi: 10.1145/2103799.2103803. https://doi.org/10.1145/2103799.2103803. (cited on page 19)

Halderman, J. A. and Teague, V., 2015. The new south wales ivote system: Security failures and verification flaws in a live online election. In *E-Voting and Identity*, 35–53. Springer International Publishing, Cham. (cited on pages 2 and 17)

Hales, T. C.; Adams, M.; Bauer, G.; Dang, D. T.; Harrison, J.; Hoang, T. L.; Kaliszyk, C.; Magron, V.; McLaughlin, S.; Nguyen, T. T.; Nguyen, T. Q.; Nipkow, T.; Obua, S.; Pleso, J.; Rute, J.; Solovyev, A.; Ta, A. H. T.; Tran, T. N.; Trieu, D. T.; Urban, J.; Vu, K. K.; and Zumkeller, R., 2015. A formal proof of the kepler conjecture. *arXiv*, 1501.02155 (2015). (cited on page 19)

Harrison, J., 1996. Hol light: A tutorial introduction. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, FMCAD '96, 265–269. Springer-Verlag, London, UK, UK. http://dl.acm.org/citation.cfm?id=646184.682934. (cited on page 27)

Harrison, J., 2006. Towards self-verification of HOL Light. In *Automated Reasoning*, 177–191. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 42)

Helios, 2016. The helios voting system. Http://heliosvoting.org/, accessed June 25, 2016. (cited on pages 8 and 123)

Hirt, M. and Sako, K., 2000. Efficient receipt-free voting based on homomorphic encryption. In *Proc. EUROCRYPT 2000*, vol. 1807 of *Lecture Notes in Computer Science*, 539–556. Springer. (cited on page 86)

Jackson, D., 2002. Alloy: A lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11, 2 (Apr. 2002), 256–290. doi:10.1145/505145.505149. https://doi.org/10.1145/505145.505149. (cited on page 9)

Jacobs, B. and Pieters, W., 2009. *Electronic Voting in the Netherlands: From Early Adoption to Early Abolishment*, 121–144. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-642-03829-7. doi:10.1007/978-3-642-03829-7_4. https://doi.org/10.1007/978-3-642-03829-7_4. (cited on page 16)

Jonker, H.; Mauw, S.; and Pang, J., 2013. Privacy and verifiability in voting systems: Methods, developments and trends. *Computer Science Review*, 10

(2013), 1–30. doi:https://doi.org/10.1016/j.cosrev.2013.08.002. https://www.sciencedirect.com/science/article/pii/S1574013713000282. (cited on page 25)

Jourdan, J.-H.; Pottier, F.; and Leroy, X., 2012. Validating lr(1) parsers. In *Programming Languages and Systems*, 397–416. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 137)

Kaufmann, M. and Strother Moore, J., 1996. Acl2: an industrial strength version of nqthm. In *Proceedings of 11th Annual Conference on Computer Assurance. COMPASS '96*, 23–34. doi:10.1109/CMPASS.1996.507872. (cited on page 27)

Klein, G.; Elphinstone, K.; Heiser, G.; Andronick, J.; Cock, D.; Derrin, P.; Elkaduwe, D.; Engelhardt, K.; Kolanski, R.; Norrish, M.; Sewell, T.; Tuch, H.; and Winwood, S., 2009. sel4: formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, 207–220. doi:10.1145/1629575.1629596. https://doi.org/10.1145/1629575.1629596. (cited on page 19)

Kohno, T.; Stubblefield, A.; Rubin, A. D.; and Wallach, D. S., 2004. Analysis of an electronic voting system. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, 27–40. doi:10.1109/SECPRI.2004.1301313. (cited on page 2)

Kremer, S. and Ryan, M., 2005. Analysis of an electronic voting protocol in the applied pi calculus. In *Programming Languages and Systems*, 186–200. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 8)

Kumar, R.; Myreen, M. O.; Norrish, M.; and Owens, S., 2014. Cakeml: a verified implementation of ML. In *Proc. POPL 2014*, 179–192. ACM. (cited on pages 9, 19, 83, and 138)

Küsters, R.; Truderung, T.; and Vogt, A., 2011. Verifiability, privacy, and coercion-resistance: New insights from a case study. In *2011 IEEE Symposium on Security and Privacy*, 538–553. doi:10.1109/SP.2011.21. (cited on pages 2 and 75)

Landin, P. J., 1964. The mechanical evaluation of expressions. *The Computer Journal*, 6, 4 (1964), 308. (cited on page 77)

Leroy, X., 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*

2006, Charleston, South Carolina, USA, January 11-13, 2006, 42–54. doi: 10.1145/1111037.1111042. https://doi.org/10.1145/1111037.1111042. (cited on pages 13, 18, and 19)

Letouzey, P., 2003. A new extraction for Coq. In *Proc. TYPES 2002*, vol. 2646 of *Lecture Notes in Computer Science*, 200–219. Springer. (cited on page 103)

Letouzey, P., 2008. Extraction in Coq: An overview. In *Proc. CiE 2008*, vol. 5028 of *Lecture Notes in Computer Science*, 359–369. Springer. (cited on page 34)

Lewis, S. J.; Pereira, O.; and Teague, V. Ceci n'est pas une preuve. https://people.eng.unimelb.edu.au/vjteague/UniversalVerifiabilitySwissPost.pdf. Accessed on October 17, 2019. (cited on pages 2 and 17)

Locher, P. and Haenni, R., 2014. A lightweight implementation of a shuffle proof for electronic voting systems. In *44. Jahrestagung der Gesellschaft für Informatik, Informatik 2014, Big Data - Komplexität meistern, 22.-26. September 2014 in Stuttgart, Deutschland*, 1391–1400. https://dl.gi.de/20.500.12116/2747. (cited on pages 7 and 104)

Marky, K.; Kulyk, O.; and Volkamer, M., 2018. Comparative usability evaluation of cast-as-intended verification approaches in internet voting. In *SICHERHEIT 2018*, 197–208. Gesellschaft für Informatik e.V., Bonn. doi:10.18420/sicherheit2018_15. (cited on page 3)

McConnell, R.; Mehlhorn, K.; Näher, S.; and Schweitzer, P., 2011. Certifying algorithms. *Computer Science Review*, 5, 2 (2011), 119 – 161. doi: https://doi.org/10.1016/j.cosrev.2010.09.009. http://www.sciencedirect.com/science/article/pii/S1574013710000560. (cited on page 22)

Mehlhorn, K. and Näher, S., 1995. Leda: A platform for combinatorial and geometric computing. *Commun. ACM*, 38, 1 (Jan. 1995), 96–102. doi: 10.1145/204865.204889. http://doi.acm.org/10.1145/204865.204889. (cited on page 23)

Meier, S.; Schmidt, B.; Cremers, C.; and Basin, D., 2013. The tamarin prover for the symbolic analysis of security protocols. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044*, CAV 2013 (Saint Petersburg, Russia, 2013), 696–701. Springer-Verlag, Berlin, Heidelberg. (cited on page 8)

Menezes, A. J.; Van Oorschot, P. C.; and Vanstone, S. A., 2018. *Handbook of applied cryptography*. CRC press. (cited on page 45)

MENEZES, A. J.; VANSTONE, S. A.; AND OORSCHOT, P. C. V., 1996. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edn. ISBN 0849385237. (cited on page 54)

MILLER, B. P.; FREDRIKSEN, L.; AND SO, B., 1990. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33, 12 (Dec. 1990), 32–44. doi:10.1145/96267.96279. http://doi.acm.org/10.1145/96267.96279. (cited on page 19)

MILNER, R., 1972. Implementation and applications of scott's logic for computable functions. *SIGPLAN Not.*, 7, 1 (Jan. 1972), 1–6. doi:10.1145/942578.807067. http://doi.acm.org/10.1145/942578.807067. (cited on page 27)

MILNER, R., 1999. *Communicating and Mobile Systems: The -Calculus*. Cambridge University Press, USA. ISBN 0521658691. (cited on page 8)

MOORE, J. S., 2019. Milestones from the pure lisp theorem prover to acl2. *Formal Aspects of Computing*, (Jul 2019). doi:10.1007/s00165-019-00490-3. https://doi.org/10.1007/s00165-019-00490-3. (cited on page 20)

MUÑOZ, C.; NARKAWICZ, A.; AND DUTLE, A., 2018. From formal requirements to highly assured software for unmanned aircraft systems. In *Formal Methods*, 647–652. Springer International Publishing, Cham. (cited on page 18)

MYREEN, M. O. AND DAVIS, J., 2014. The reflective milawa theorem prover is sound - (down to the machine code that runs it). In *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, 421–436. doi:10.1007/978-3-319-08970-6_27. https://doi.org/10.1007/978-3-319-08970-6_27. (cited on page 19)

NAMI, M. AND SURYN, W., 2013. Software testing is necessary but not sufficient for software trustworthiness. In *Trustworthy Computing and Services*, 34–44. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 18)

NIPKOW, T.; PAULSON, L. C.; AND WENZEL, M., 2002a. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, vol. 2283 of *Lect. Notes in Comp. Sci.* Springer. (cited on page 27)

NIPKOW, T.; WENZEL, M.; AND PAULSON, L. C., 2002b. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg. ISBN 3-540-43376-7. (cited on page 24)

NORELL, U., 2009. Dependently typed programming in agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08

(Heijen, The Netherlands, 2009), 230–266. Springer-Verlag, Berlin, Heidelberg. http://dl.acm.org/citation.cfm?id=1813347.1813352. (cited on page 27)

Otten, J., 2003. Fuller disclosure than intended. (2003). http://www.votingmatters.org.uk/ISSUE17/I17P2.PDF. Accessed on October 17, 2019. (cited on pages 9, 25, 82, and 85)

Owre, S.; Rushby, J. M.; and Shankar, N., 1992. Pvs: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*, CADE-11, 748–752. Springer-Verlag, London, UK, UK. http://dl.acm.org/citation.cfm?id=648230.752639. (cited on page 27)

Paar, C. and Pelzl, J., 2009. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Publishing Company, Incorporated, 1st edn. ISBN 3642041000, 9783642041006. (cited on page 54)

Paillier, P., 1999. Public-key cryptosystems based on composite degree residuosity classes. In *International conference on the theory and applications of cryptographic techniques*, 223–238. Springer. (cited on page 47)

Pattinson, D. and Schürmann, C., 2015. Vote counting as mathematical proof. In *Proc. AI 2015*, vol. 9457 of *Lecture Notes in Computer Science*, 464–475. Springer. (cited on pages 8, 9, and 22)

Paulin-Mohring, C., 1993. Inductive definitions in the system Coq - rules and properties. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, TLCA '93, 328–345. Springer-Verlag, London, UK, UK. http://dl.acm.org/citation.cfm?id=645891.671440. (cited on pages 28 and 32)

Pedersen, T. P., 1992. Non-interactive and information-theoretic secure verifiable secret sharing. In *Advances in Cryptology — CRYPTO '91*, 129–140. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on pages 92, 112, and 116)

Pfenning, F. and Paulin-Mohring, C., 1989. Inductively defined types in the calculus of constructions. In *International Conference on Mathematical Foundations of Programming Semantics*, 209–228. Springer. (cited on page 32)

Pierce, B. C., 2004. *Advanced Topics in Types and Programming Languages*. The MIT Press. ISBN 0262162288. (cited on page 35)

POHLIG, S. AND HELLMAN, M., 2006. An improved algorithm for computing logarithms over gf(p) and its cryptographic significance (corresp.). *IEEE Trans. Inf. Theor.*, 24, 1 (Sep. 2006), 106–110. doi:10.1109/TIT.1978.1055817. https://doi.org/10.1109/TIT.1978.1055817. (cited on page 48)

POLLACK, R., 1998. How to believe a machine-checked proof. *Twenty Five Years of Constructive Type Theory*, 36 (1998), 205. (cited on page 41)

RIVEST, R. L., 2008. On the notion of software independence in voting systems. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366, 1881 (2008), 3759–3767. https://royalsocietypublishing.org/doi/10.1098/rsta.2008.0149. (cited on page 21)

RIVEST, R. L.; ADLEMAN, L.; DERTOUZOS, M. L.; ET AL., 1978. On data banks and privacy homomorphisms. (1978). (cited on page 46)

RIVEST, R. L. AND SHEN, E., 2010. An optimal single-winner preferential voting system based on game theory. In *Proc. COMSOC 2010*. Duesseldorf University Press. (cited on page 57)

RYAN, P. Y. A.; RØNNE, P. B.; AND IOVINO, V., 2016. Selene: Voting with transparent verifiability and coercion-mitigation. In *Financial Cryptography and Data Security*, 176–192. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 8)

SCHNEIER, B., 1995. *Applied Cryptography (2Nd Ed.): Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., New York, NY, USA. ISBN 0-471-11709-9. (cited on page 54)

SCHNORR, C. P., 1990. Efficient identification and signatures for smart cards. In *Advances in Cryptology — CRYPTO' 89 Proceedings*, 239–252. Springer New York, New York, NY. (cited on pages 51 and 124)

SCHULZE, M., 2011. A new monotonic, clone-independent, reversal symmetric, and Condorcet-consistent single-winner election method. *Social Choice and Welfare*, 36, 2 (2011), 267–303. (cited on pages 4, 6, 56, and 70)

SCHÜRMANN, C., 2009. Electronic elections: Trust through engineering. In *Proc. RE-VOTE 2009*, 38–46. IEEE Computer Society. (cited on page 22)

SLIND, K. AND NORRISH, M., 2008. A brief overview of Hol4. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '08 (Montreal, P.Q., Canada, 2008), 28–32. Springer-Verlag, Berlin, Heidelberg. doi:10.1007/978-3-540-71067-7_6. http://dx.doi.org/10.1007/978-3-540-71067-7_6. (cited on pages 9 and 27)

SOZEAU, M. AND MANGIN, C., 2019. Equations reloaded: High-level dependently-typed functional programming and proving in Coq. *Proc. ACM Program. Lang.*, 3, ICFP (Jul. 2019), 86:1–86:29. doi:10.1145/3341690. http://doi.acm.org/10.1145/3341690. (cited on page 116)

STANDARD, D. E. ET AL., 1999. Data encryption standard. *Federal Information Processing Standards Publication*, (1999), 112. (cited on page 42)

STOLTENBERG-HANSEN, V.; LINDSTRÖM, I.; AND GRIFFOR, E., 1994. *Mathematical Theory of Domains*. No. 22 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press. (cited on page 65)

SULLIVAN, G. F. AND MASSON, G. M., 1990. Using certification trails to achieve software fault tolerance. In *[1990] Digest of Papers. Fault-Tolerant Computing: 20th International Symposium*, 423–431. doi:10.1109/FTCS.1990.89397. (cited on page 22)

TARSKI, A., 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5, 2 (1955), 285—309. (cited on page 65)

TERELIUS, B. AND WIKSTRÖM, D., 2010. Proofs of restricted shuffles. In *AFRICACRYPT*, vol. 6055 of *Lecture Notes in Computer Science*, 100–113. Springer. (cited on page 90)

VERITY, F. AND PATTINSON, D., 2017. Formally verified invariants of vote counting schemes. In *Proceedings of the Australasian Computer Science Week Multiconference*, ACSW '17 (Geelong, Australia, 2017), 31:1–31:10. ACM, New York, NY, USA. doi:10.1145/3014812.3014845. http://doi.acm.org/10.1145/3014812.3014845. (cited on page 8)

VERITY, F.; PATTINSON, D.; AND GORÉ, R. Modular synthesis of provably correct vote counting programs. 55. (cited on page 8)

WIKSTRÖM, D., 2009. A commitment-consistent proof of a shuffle. In *Proceedings of the 14th Australasian Conference on Information Security and Privacy*, ACISP '09 (Brisbane, Australia, 2009), 407–421. Springer-Verlag, Berlin, Heidelberg. doi:10.1007/978-3-642-02620-1_28. http://dx.doi.org/10.1007/978-3-642-02620-1_28. (cited on pages 92, 93, 112, and 116)

WILCOX, J. R.; WOOS, D.; PANCHEKHA, P.; TATLOCK, Z.; WANG, X.; ERNST, M. D.; AND ANDERSON, T. E., 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, 357–368. doi:10.1145/2737924.2737958. https://doi.org/10.1145/2737924.2737958. (cited on page 19)

Wolchok, S.; Wustrow, E.; Halderman, J. A.; Prasad, H. K.; Kankipati, A.; Sakhamuri, S. K.; Yagati, V.; and Gonggrijp, R., 2010. Security analysis of India's electronic voting machines. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10 (Chicago, Illinois, USA, 2010), 1–14. ACM, New York, NY, USA. doi:10.1145/1866307.1866309. http://doi.acm.org/10.1145/1866307.1866309. (cited on pages 2 and 16)

Yang, X.; Chen, Y.; Eide, E.; and Regehr, J., 2011. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46, 6 (Jun. 2011), 283–294. doi:10.1145/1993316.1993532. http://doi.acm.org/10.1145/1993316.1993532. (cited on page 19)

Zhao, J. and Zdancewic, S., 2012. Mechanized verification of computing dominators for formalizing compilers. In *Certified Programs and Proofs*, 27–42. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 19)