Formally Verified Electronic Voting Scheme : A Case Study

Mukesh Tiwari

A thesis submitted for the degree of
YOUR DEGREE NAME
The Australian National University

October 2019

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

Mukesh Tiwari
9 October 2019

to my xxx, yyy (yyy is the people you want to dedicated this thesis to.)

Acknowledgments

This thesis could not have been possible without the support of my supervisor, Dirk Pattinson. I really admire his ability to understand the problem, and his intuition to to make sure that I stay clear from many dead ends which I would have happily spent months. I wish I could incorporate more of his qualities, but I believe I have less optimism about my chance.

Abstract

Put your abstract here.

Contents

List of Figures

List of Tables

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

Electronic voting is a nightmare because the minuscule possibility of a bug in software used in voting could lead to a disaster, possibly inverting the results [1] [Wolchok et al., 2010], [Halderman and Teague, 2015], [Aranha et al., 2019], [Feldman et al., 2007]. Given that the risk of electronic voting is so high, we should totally refrain from it. However, electronic voting is getting popular in many countries despite the fact that electronic voting has many undesirable problems. The reason that electronic voting is getting popularity in many countries is cost-effective, increase in voter turnout and faster result. India has 900 million eligible voter, and with 67 percent (roughly 600 million) voter turnout in 2019 general election, the result was declared in 2 days. This was possible because India uses electronic voting machines (EVM) to conduct the election. Australia has compulsory voting, but its massive size with sparsely populated land makes the election a big logistic challenge. Australia is actively pursuing Internet voting to ease the logistic challenges and engage more citizens from the remote places. Estonia is praised for successfully implementing the Internet voting since 2005. The first country to adopt internet voting and showing confidence in technology. In 2019 elections, as claimed on e-estonia [2], the time saved in was 11,000 working days.

It seems that electronic voting is a silver bullet and solution to many problems, and we can safely ignore the problems posed by it. It is a worthwhile discussion that what a democracy want? A faster, but wrong and unverifiable result, or a slower, but correct and verifiable result. By its inherent nature, electronic voting has many

[1] https://people.eng.unimelb.edu.au/vjteague/UniversalVerifiabilitySwissPost.pdf

[2] https://e-estonia.com/solutions/e-governance/i-voting/

problems, which are not present in paper ballot election, that makes it perfectly susceptible to delivering wrong and unverifiable result. Because of the untrusted and complex software [3] and hardware stack [4], electronic voting lacks:

1. correctness, i.e. produced results are correct and convincing to everyone leaving no ground for suspicion.
2. privacy, i.e. protecting the identity of the voter of a cast ballot and the choices on the cast ballot.
3. verification, i.e. treated as a black-box which violates the fundamental property of public examinability.

These untrusted components exposes a large attack surface which could potentially be exploited for illegal gain in election, possibly by current government or foreign country. As a result, a man or women who have not legitimately received a majority or quota would be put in charge of public office. In a nutshell, electronic voting violates fundamental principals of any democratic election, i.e. correctness, privacy, and verifiability.

1.2 Research Motivation

This thesis sets out to answer the question of correctness, privacy and verifiability in electronic voting. The questions we asked ourself was:

1. Can we implement a vote counting protocol with the guaranteed correctness of its properties, and practical enough to count the real life election involving millions of ballots ? (Correctness and Usability)
2. Can we produce the result by counting encrypted ballot 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)
3. Can we decouple the verifiability from implementation, i.e. generating enough evidence so that any independent auditor can ascertain the outcome of election without trusting the implementation of software used to conduct the election ? (Verifiability)

We achieve the correctness by using formal verification, privacy using encryption, and verifiability using scrutiny sheet.

A democracy can be best describe as a system where every participant has equal right to express his opinion(s) on different matters. Each eligible participant expresses his opinion, and a winner is elected by using some method to combine every participant's opinion. In current context of democratic elections, every eligible

[3] Unix operating system has 15 million code which can not trusted at all

[4] Intel has many undocumented instructions in its x86 processor. https://github.com/xoreaxeaxeax/sandsifter

participant expresses his opinion on a paper, also known as ballot, by marking a tick against the preferred candidate (first-past-the-post) or ranking the candidates in some order (preferential voting). One of the most fundamental assumption during any democratic election, privacy and anonymity about the voter's choices, was not the case in past. It was Victoria, Australia who first introduce the concept of secret ballot in 1856 [5] [6] and gradually, it got adoption all over the world. Before 1856, Victoria and NSW held their elections to elect its democratic representative in pub where it was legal for candidates to offer beer to voters to influence their decision!

Verifiability or transparency is other fundamental assumption for any democratic election. Transparency insures that each step of of election process is easily understood and can be scrutinize by every stake holders e.g. voters, political parties, and external observers. Privacy and verifiability are basic ingredient for coercion free election, i.e. people can express their opinion with any concern, and at the same time, they can not sell their votes. Without privacy and verifiability, we can not expect a true democracy.

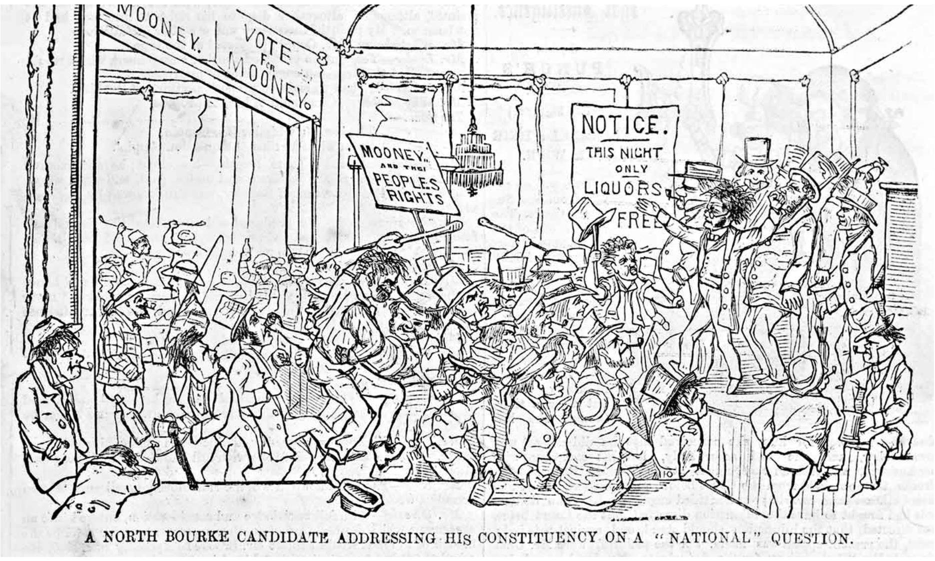

MOONEY.
VOTE For MOONEY.
MOONEY. AND THE PEOPLES RIGHTS
NOTICE. THIS NIGHT ONLY LIQUORS FREE
A NORTH BOURKE CANDIDATE ADDRESSING HIS CONSTITUENCY ON A "NATIONAL" QUESTION.

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

Electronic voting is very different from paper ballot election. Reasoning about privacy and verifiability and many other properties in paper ballot election is straight forward, but we can not do the same for electronic voting election. The reason is, electronic voting constitutes of various untrusted component of software [7] and hardware [8] which exposes a large attack surface. These attack surfaces could potentially

[5] https://trove.nla.gov.au/work/25532676?q&versionId=30761831

[6] https://www.nma.gov.au/defining-moments/ resources/secret-ballot-introduced

[7] Unix operating system has 15 million code which can not trusted at all

[8] Intel has many undocumented instructions in its x86 processor. https://github.com/xoreaxeaxeax/sandsifter

be exploited for illegal gain in election. Even though electronic voting is gaining popularity, none so far has addressed the software bug issue in electronic voting context. Most of the software used in electronic voting process are inadequately tested which evidently can not rule out all the possibilities.

1.3 Motivation and Research Statement

Motivation: Electronic voting has gained a lot of attention in recent year, and there are extensive work which addresses the different issues related of electronic voting protocols in symbolic model. Kremer and Ryan [2005], and Delaune et al. [2010a] have used pi-calculus to model and analyse various properties, fairness, eligibility, vote-privacy, receipt-freeness and coercion-resistant, of the protocol FOO developed by Fujioka et al. [1993]. Backes et al. [2008] presented a general technique to model remote electronic voting protocol and automatically verifying its security properties using pi-calculus. Cortier and Smyth [2011] have used pi-calculus to analyse the ballot secrecy of Helios [2016]. Cortier and Wiedling [2012] have used pi-calculus to ascertain the properties of Norwegian electronic voting protocol. Bruni et al. [2017] have used Tamarin to prove receipt-freeness and vote-privacy of Selene voting protocol [Ryan et al., 2016]. Most of these work differs from ours in the sense that their primary focus is verification of security protocol in Dolev-Yao model or complexity-theoretic model, whereas our work is more focused on verified implementation and verifiability aspect of election. [Note to myself: Carsten has used the term "auditable correctness of implementations" for this]

The closest to our work is DeYoung and Schürmann [2012], Pattinson and Schürmann [2015], Pattinson and Verity [2016], Verity and Pattinson [2017], and Ghale et al. [2017]. DeYoung and Schürmann [2012] used linear logic[Girard, 1987] to model the different entity 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 he might need to show his photo id to multiple times at counting booth. Pattinson and Schürmann [2015] treated the process of vote counting from the perspective of mathematical proof. They used (mathematical) proof theory to model the counting. Ghale et al. [2017] have formalized the single transferable votes in Coq and extracted a Haskell code from the formalization. The extracted Haskell code produces the result and a certificate for a given set of input ballots. The certificate can be used by any third party to verify or audit the outcome of election result. However, none of these work considers privacy 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].

This work not only considers the plaintext ballot, but extends the counting to encrypted ballot using homomorphic encryption to preserve the privacy of election. To ensure the verifiability, we use the zero-knowledge-proof to make sure that outcome of election can be attested by any independent third party. To ease the auditing or certificate-checking of election conducted on encrypted ballot with other complex cryptographic entities, we have developed a formally verified certificate checker for auditing the election [9]. To the best of our knowledge, no one has developed a vote counting system which counts encrypted ballots, and at the same time, the implementation itself is verified inside a theorem prover. [Ghale et al., 2018] has developed a verified certificate checker, but, again, it involves plaintext ballots with simple mathematics of addition and multiplication while our checker tackles the certificates involving complex cryptographic objects, e.g. encrypted ballots and zero-knowledge-proof of different statements.

Research Statement: After inspecting the current state of art in electronic voting, we asked ourselves three questions:

1. Can we engineer a system which is formally verified to guarantee the correctness property, and practical enough to count the real life election involving millions of ballots ?

2. Can we decouple the verifiability from implementation, i.e. generating enough evidence so that any independent auditor can ascertain the outcome of election without trusting the implementation of software used to conduct the election ?

3. How "good" is our proposed solution ?

In order to answer these three questions, we needed three things:

1. A voting protocol

2. A theorem prover to implement the voting protocol

3. Parameters, agreed upon properties in electronic voting research community, against which we will measure our solution

1.3.1 Voting Protocol

For voting protocol, we settled down with Schulze [2011]. Even though Schulze method is not used in any democratic election, the reason we went ahead with it because it is interesting and at the same time, non-trivial. Schulze's method elects a single winner based on preferential votes. At the same time, Arrow's impossibility theorem [Arrow, 1950] states that no preferential voting scheme can have all the desired properties established by social choice theorist, the Schulze's method offers a good balance. Many of these properties are already established in his original paper. We will discuss more about Schulze method in 3, and about its properties in 6.

1.3.2 Theorem Prover:

Our preference for theorem prover was Coq [Bertot et al., 2004]. The reason for choosing Coq is that it supports an expressive logic and (crucial for us) dependent

[9] IACR 2018 election

inductive types. 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 election to produce the result. We will discuss more about the Coq in 2.

1.3.3 Properties of Electronic Voting Protocol:

What makes a electronic voting protocol "good" or "desirable" ? Some commonly sought properties which a electronic voting protocol must have are given below [Küsters et al., 2011], [Benaloh and Tuinstra, 1994], [Delaune et al., 2010b], [Bernhard et al., 2017].

- Correctness: The produced results are correct, and convincing to all leaving no ground for suspicion.
- Privacy: All the votes must be secret, and voter should not be able to convince anyone the value of his 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-category:
  - 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 final result on the basis of the collected ballots.
- Practicality/Usability:
- Coercion-resistance: A voter cannot cooperate with a coercer to prove to him that she voted in a certain way.
- Voter Eligibility: Only eligible voter can cast the ballot, and only once.
- Fairness: No intermediate results can be obtained which could influence the remaining voters.
- Receipt-freeness: A voter does not gain any information (receipt) which can be used to prove to a coercer that she voted in a certain way

The focus of our study is the *Correctness*, *Practicality*, *Verifiability*, i.e. generating enough evidence so that everyone can verify final result on the basis of the collected ballots, and *Privacy*. In the critical summary of each chapter, we will measure our solution against these properties.

1.4 Publication

During the course of PhD, I was fortunate to work with many researchers including my supervisor Dr. Dirk Pattinson, co-supervisor Rajeev Goré, colleague Milad K. Ghale, and external collaborators Dr. Thomas Haines from NTNU, Dr. Lyria Bennett Moses from UNSW, and Dr Ron Levy from ANU.

1. Pattinson, D. and Tiwari, M., 2017. Schulze voting as evidence carrying computation. In Proc. ITP 2017, vol. 10499 of Lecture Notes in Computer Science, 410–426. Springer.

2. Lyria Bennett Moses, Rajeev Goré, Ron Levy, Dirk Pattinson, Mukesh Tiwari: No More Excuses: Automated Synthesis of Practical and Verifiable Vote-Counting Programs for Complex Voting Schemes. E-VOTE-ID 2017: 66-83

3. Milad K. Ghale, Rajeev Goré, Dirk Pattinson, Mukesh Tiwari: Modular Formalisation and Verification of STV Algorithms. E-Vote-ID 2018: 51-66

4. VSSTE paper

5. CCS paper

1.5 Outline of the Chapters

[Rewrite again when you finish everything] In chapter 2, we will give a brief history of electronic voting, the glitches that cause some countries to withdraw from electronic voting.

We will evaluate our work based on the agreed upon properties of electronic voting in the research community. It is privacy, verifiability and practicality. In the end of each chapter, we will give a critical summary of our work and which parameter is passed successfully and which one it failed.

How many chapters you have? You may have Chapter 2, Chapter **??**, Chapter **??**, Chapter **??**, and Chapter **??**.

Chapter 2

Background

> People shouldn't be afraid of their government. Governments should be afraid of their people..
>
> *Alan Moore, V for Vendetta*

Electronic voting is a nightmare because the minuscule possibility of a bug in software used in voting could lead to a disaster, possibly inverting the results [1]. Given that the risk of electronic voting is so high, we should totally refrain from it; however, it is gaining popularity. Some of the notable countries using some form of electronic voting are Estonia (probably the only success story), India, Australia, Canada and USA. The world can be divided into five broad categories according to the usage of electronic voting[2] 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: Write here a chapter outline

2.1 Electronic Voting

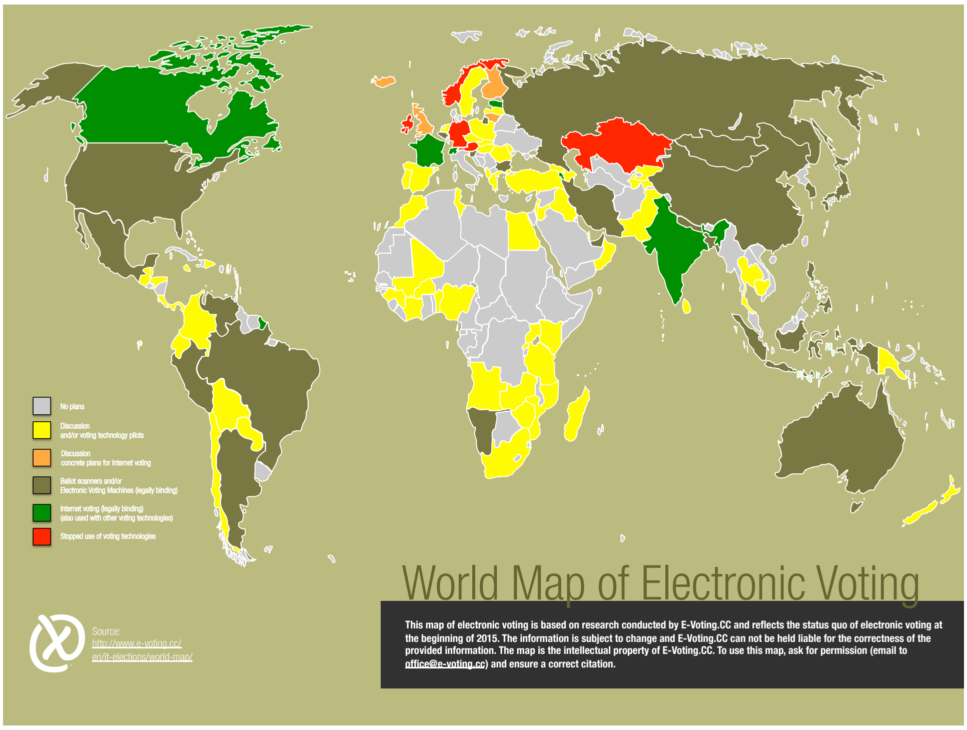
Electronic voting is already happening, and the arguments in favour of electronic voting include increased voter turnout, faster result and lower cost. The Senate election conducted in Western Australia in September 2013 was declared void by the high court because of the loss of 1370 votes. It was re-conducted in April 2014 with the cost of 20 Million AUD with additional delay in results[3]. Sometimes, not only cost, but time as well is a major concern. . The advantages of electronic voting look so promising, so why have some countries (Red Area) retracted from electronic vot-

Give a example of seat which took considerable amount of time in declaring result. I can't find it :(

[1] https://people.eng.unimelb.edu.au/vjteague/UniversalVerifiabilitySwissPost.pdf

[2] https://www.e-voting.cc/en/it-elections/world-map

[3] https://www.theguardian.com/world/2014/feb/28/western-australia-senate-election-re-run-to-be-held-on-5-april


⬜ No plans
🟨 Discussion and/or voting technology pilots
🟧 Discussion concrete plans for Internet voting
▨ Ballot scanners and/or Electronic Voting Machines (legally binding)
🟩 Internet voting (legally binding) (also used with other voting technologies)
🟥 Stopped use of voting technologies

World Map of Electronic Voting

Source: http://www.e-voting.cc/en/it-elections/world-map/

This map of electronic voting is based on research conducted by E-Voting.CC and reflects the status quo of electronic voting at the beginning of 2015. The information is subject to change and E-Voting.CC can not be held liable for the correctness of the provided information. The map is the intellectual property of E-Voting.CC. To use this map, ask for permission (email to office@e-voting.cc) and ensure a correct citation.

Figure 2.1: World map of Electronic Voting

ing? Electronic voting makes the process faster, but it has its own layer of added complexities which creates trust issues amongst voters.

Germany: In the 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, and these machines could be hacked, hence results of the 2005 election could not be trusted. The case was argued on the grounds of "all essential steps in the elections are subject to public examinability" according to German Constitution (Basic Law for the Federal Republic of Germany). The Court noted that, under the constitution, elections are required to be public in nature:

"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." [4]

The court did not rule out or prevent the usage of electronic voting machines, but suggested to make the process more transparent and trustworthy.

"The legislature is not prevented from using electronic voting machines in the elections if

[4] https://www.bundesverfassungsgericht.de/ SharedDocs/Entscheidungen/EN/2009/03/cs20090303_2bvc000307en.html

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 were 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. The security flaw was demonstrated by Dutch public foundation, Wij vertrouwen stemcomputers niet [5]. They showed that e-voting machines used in election leaks enough information to guess the option or choice of a voter, and they can be easily intercepted from 20 to 30 meters [6].

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 it. It was already shown in 2010 in the paper Security Analysis of India's Electronic Voting Machines by Wolchok et al. [2010] that it is possible to manipulate the election results by replacing the parts of machine with malicious look alike components along with sending them instructions over wireless [7] [8].

Australia In March, 2015 state election of New South Wales, Australia, a Internet voting system, iVote, was used and 280,000 votes were cast through it. NSW Election commissioner claimed that it was *"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."* [9]

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

[5] English Translation "We do not trust voting computers"

[6] https://www.youtube.com/watch?v=B05wPomCjEY

[7] https://www.youtube.com/watch?v=ZlCOj1dElDY

[8] https://indiaevm.org/

[9] https://www.abc.net.au/news/2015-02-04/computer-voting-may-feature-in-march-nsw-election/6068290

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 a online election for 6 days on buggy software which was susceptible to many attacks with a possible outcome of tampered ballot without anyone noticing it.

There are various factors for these debacles, but one of the most common denominator among all these debacles which contributed significantly is the software used in the election process. These software are developed via a process called software engineering process or software development process, and it varies from company to company, but roughly, it translates to requirement gathering, software design, implementation, testing, and maintenance. During this whole process of transforming a vague idea into concrete software, testing the is the only phase which deals with software bugs. However, testing is not adequate measure for instilling the confidence in software that it is bug free as stated by Edsger W. Dijkstra: *"Program testing can be used to show the presence of bugs, but never to show their absence!"*. Besides, these software are closely guarded secrets and their source code in not open for general public because of commercial interests of companies [10].

In the next section, I will discuss that software testing in not adequate for electronic voting software, and we should prove the correctness of software by using formal verification techniques Beckert et al. [2014]. I will argue that having a rigorous software development methodology would alleviate the bug problem with few case studies 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.

2.1.1 Software Bugs : A Formal Method Approach

Formal verification has been successfully applied in many areas ranging from verified C compilers CompCert[reference], verified ML compiler CakeML, Vellvm: Verifying the LLVM, verified cryptography Fiat-crypto, verified operating system CertiKOS and SeL4, verified theorem prover Milwa, verified crash resistant file system FSCQ, verified distributed system Verdi, mechanisation of Four Color Theorem, Fundamental Theorem of Algebra, and Kepler Conjecture. One thing I would like to emphasize is that none of these are toy project, and it takes years to develop and verify them e.g. SeL4 took 5 years [find the time and cost of these projects]. Also, some of these products are used in commercially e.g CompCert is used by AIRBUS, and MTU [11]

[10] https://www.aec.gov.au/information-access/foi/2014/files/ls4912-1.pdf
[11] https://www.absint.com/compcert/

and Fiat-crypto is used in Google's BoringSSL library for elliptic-curve arithmetic [12]. The very basic question to ponder about using formal method to develop software: i) Does it achieve the goal given the cost and effort ? ii) Are these software bug free ?

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 [13] and LLVM [14]; however, they could not find any bug in 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."

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 proverto 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 handy, 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't afford to lose a single ballot or miscalculation or any undefined behaviour, should be developed rigorously using formal method techniques.

2.1.2 Verification and Verifiability

Formal verification is useful in producing the bug free code, but it does not answer the question that why should a voter trust the formally verified system. We solely can not establish the trust in the system based on argument of formal verification. I would call formal verification a necessary, but not the sufficient condition. Combing both *verification* of the computer software that counts votes, and *verifiability* of individual counts are critical for building trust in an election process. Given the mission-

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

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

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

critical importance of correctness of vote-counting, both for the legal integrity of the process and for building public trust, it is imperative to replace the currently used black-box software for vote-counting with a counterpart that is both verified and produces evidence, we call it certificate or scrutiny sheet, which can later be used to certify the outcome of election.

In order to ascertain that the results of a verified program are indeed correct, 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?

3. ensure that the computing equipment on which the (verified) program is executed has not been tampered with or is otherwise compromised, and finally

4. ascertain that it was indeed the verified program that was executed in order to obtain the claimed results.

The trust in correctness of any result rests on all items above. The second two items are more problematic as they require trust in the integrity of equipment, and individuals, both of which can be hard to ascertain once the computation has completed. The first two trust requirements can be met by publishing both the specification and the correctness proof so that the specification can be analysed, and the proof can be replayed. Both need a considerable amount of expertise but can be carried out by (ideally more than one group of) domain experts. Trust in the correctness of the result can still be achieved if a large enough number of domain experts manage to replicate the computation, using equipment they know is not compromised, and running the program they know has been verified. As such, trust in *verified* computation mainly rests on a relatively small number of domain experts.

2.1.3 Legal Aspects of Verification and Verifiability

Any system for counting votes in democratic elections needs to satisfy at least three conditions: (1) each person's vote must be counted accurately, according to a mandated procedure, (2) the system and process should be subjectively trusted by the electorate, (3) there should be an objective basis for such trust, or in other words the system must be trustworthy. While subjective trust cannot be guaranteed through greater transparency O'Neill [2002], transparency about both the voting system and the actual counting of the vote in a particular election are important in reducing errors and ensuring an accurate count, promoting public trust and providing the evidential basis for demonstrated trustworthiness. In particular, it is a lack of transparency that has been the primary source of criticism of existing systems, both in the literature Carrier [2012]; Conway et al. [2017] and among civil society organisations Vogl [2012] (for example, `blackboxvoting.org` and `trustvote.org`). International commitment to transparency is also demonstrated through initiatives such as the Open Government Partnership. Another important concept referred to both in the literature and by civil society organisations is public accountability, which requires both giving an "account" or explanation to the public and when called on (for example, in court) as well as being held publicly responsible for failures. Transparency is thus a crucial component of accountability, although the latter will involve other features (such as enforcement mechanisms) that are beyond the scope of this paper.

There are two contexts in which transparency is important in the running of elections. First, there should be transparency in Hood's sense Hood [2001] as to the process used in elections generally. This is generally done through legislation with detailed provisions specifying such matters as the voting method to be used as well as the requirements for a vote to count as valid. In a semi-automated process, this requires a combination of legislation (instructions to humans) and computer code (instructions to machines). The second kind of transparency, corresponding to Meijer's use of the term Meijer [2014], is required in relation to the performance of these procedures in a specific election. In a manual process, procedural transparency is generally only to intermediaries, the scrutineers, who are able to observe the handling and tallying of ballot papers in order to monitor officials in the performance of their tasks. While the use of a limited number of intermediaries is not ideal, measures such as allowing scrutineers to be selected by candidates (eg Commonwealth Electoral Act 1918 (Australia) s 264) promote public confidence that the procedure as a whole is unbiased. However imperfect, procedural transparency reduces the risk of error and fraud in execution of the mandated procedure and enhances trust.

Electronic vote counting ought to achieve at least a similar level of transparency along both dimensions as manual systems in order to promote equivalent levels of trust. Ideally, it would go further given physical limitations (such as the number of scrutineers able to fit in a room) apply to a smaller part of the process. The use of a verified, and fully verifiable system is transparent in both senses, with members of the public able to monitor both the rules that are followed and the workings and performance of the system in a particular instance.

First, the vote counting procedure needs to be transparent. For electronic vote counting, the procedure is specified in both legislation (which authorises the electronic vote counting procedure) and in the software employed. The use of open source code ensures that the public has the same level of access to instructions given to the computer as it has to legislative commands given to election officials. The use of open source code is crucial as is demonstrated through a comparison of different jurisdictions of Australia. In Australia, for example, the Federal Senate and NSW state election vote counting are based on proprietary black box systems while the Australian Capital Territory uses open source eVACS software Australian Electoral Commission [2013]; Conway et al. [2017]; Elections ACT [2016]. This has significant impact on the ability of researchers to detect errors both in advance of elections and in time to correct results Conway et al. [2017]. Private verification systems have been less successful, in both Australia and the US, in providing equivalent protection against error to open source software Carrier [2012]; Conway et al. [2017]. Further, private verification provides a lower level of public transparency than the use of manual systems which rely on public legislation (instructions to humans) as the primary source of vote counting procedures Carrier [2012]. It should also be noted that there are few public advantages in secrecy since security is usually enhanced by adopting an open approach (unless high quality open source vote counting software were unavailable), and private profit advantages are outweighed by the importance of trust in democratic elections.

Second, verifiability provides a method of ascertaining the correctness of results of a specific election. External parties are able to check a certificate to confirm that the counting process has operated according to the rules of the voting procedure. Under a manual process, tallying and counting can only be confirmed by a small number of scrutineers directly observing human officials. The certification process allows greater transparency not limited to the number of people able to fit within a physical space, although we recognise that physical scrutiny is still required for earlier elements of the voting and vote counting process (up to verification of optical scanning of ballots). Certification reduces the risk of error and fraud that would compromise accuracy and provides an evidence-base for trustworthiness. It is also likely to increase subjective public trust, although this will require engagement with the public as to the nature of verification involved. While it is likely that in practice checking will be limited to a small group with the technical expertise, infrastructure and political interest to pursue it, knowledge as to the openness of the model is likely to increase public trust. Currently in Australia, neither open source nor proprietary vote counting systems provide an equivalent level of procedural transparency for monitoring the count in a particular election (for example, compare Commonwealth Electoral Act 1918 (Australia) s 273A).

Ultimately, legislation, computer code (where relevant) and electoral procedures need to combine to safeguard an accurate count in which the public has justified confidence. The verification and verifiability measures suggested here go further to ensure this than current methods used in Australia and, as far as we are aware, public office elections around the world.

2.1.4 Scrutiny Sheet

Scrutiny sheet is the tabulation of relevant data to ascertain 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], Certifying-algorithms[15], Arkoudas and Rinard [2005], and in Schürmann [2009] in the context of electronic voting. In general, the idea of computation is that a computable function f, it takes a input x and produces output y; 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 $f(x) = y$.

[15] https://people.mpi-inf.mpg.de/ mehlhorn/ftp/CertifyingAlgorithms.pdf

program for f

Below is simple piece of Haskell code, but the choice of language does not matter as long as it produces a certificate, which computes the greatest common divisor of two numbers with a certificate. The first component of computation is the result and second one the certificate.

```
gcdWithCertificate :: Integer -> Integer -> (Integer, String)
gcdWithCertificate a b
 | b == 0 = (a, "gcd " ++ show a ++ " 0 = " ++ show a)
 | otherwise =
   let (rgcd, rcertificate) = gcdWithCertificate b (mod a b) in
   (rgcd, "gcd " ++ show a ++ " " ++ show b ++ "\n" ++ rcertificate)
```

After running the program on the concrete input 144 and 89, it produces the result and certificate (below).

```
gcd 144 89
-------------
gcd 89 55
-------------
     .
     .
     .
-------------
gcd 2 1
```

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

In order to verify the correctness of computation of greatest common divisor (gcd), all we need to do is check that one of the rule, give below, is applicable at every stage of computation:

1. $\forall$ x, gcd x 0 = x
2. $\forall$ x y, gcd x y = gcd y (mod x y)

It is very crucial to know that the primary purpose of checkers are to verify the correctness of computation, hence, it is important that they should be: i) correct, and ii) simple. We will discuss more about certificate checking in chapter Software Independence, and we will argue that why formal verifying the checkers is a good idea to achieve the correctness. We will also present a case study of formally verified checker for *IACR 2018* election.

2.2 Summary

We reiterate that to maximise trust, reliability and auditability of electronic vote counting, we need both approaches, verification and verifiability, to be combined. To ensure (universal) verifiability, we advocate that vote-counting programs do not only compute a final result, but additionally produce an independently verifiable certificate that attests to the correctness of the computation, together with a formal verification that valid certificates indeed imply the correct determination of winners. Given a certificate-producing vote-counting program, external parties or stakeholders can then satisfy themselves to the correctness of the count by checking the certificate.

In the next chapter, I will briefly discussion about Coq theorem, and why did we choose it to formalize the Schulze method.

Can I introduce Write Hilbert's idea of mathematical formalism/Frege

A proof assistant is a computer program which assists users in development of mathematical proofs. The idea of developing mathematical proofs using computer goes back to Automath (automating mathematics) [cite Automath] and LCF [cite Logic for computation] project. 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. Automath was first practical project to exploit the Curry-Howard isomorphism (proofs-as-programs and formulas-as-types) [reference here]. DeBruijn was likely unaware of this correspondence, and he almost re-invented it ([Wiki entry on Curry-Howard]). Many researchers refers Curry-Howard isomorphism as Curry-Howard-DeBruijn isomorphism. Automath project can be seen as the precursor of proof assistants NuPrl [cite here] and Coq [cite coq]. Some other notable proof assistants are LCF (Logic for Computable Functions) [cite Milner?], Mizar [cite], Nqthm/ACl2 [cite], PVS [cite], HOL (a family of tools derived from LCF theorem prover), Agda [cite], and Lean [cite].

In this chapter, I will give a overview of theoretical foundation of Coq thorem prover i.e. Calculus of Construction and Calculus of Inductive Construction, followed by Dependent type and how it leads to correct by construction (paradigm?) with a example of dependent typed lambda calculus. I will also discuss distinction between Type and Prop and how it affects the code extraction, a feature for extracting certified functional programs from specification proofs, and the specification language Gallina. Finally, I would argue that why should we trust the Coq proofs even though it does not match or look like a mathematical proof.

2.3 Coq: Interactive Theorem prover

explain here a about Coq. What is Coq ?

The Coq proof assistant is an interactive theorem prover based on underlying theory of Calculus of Inductive Construction [Cite Pualine Mohring] which itself is an augmentation of Calculus of Construction [cite Huet and Coquand] with inductive data-type.

2.3.1 Calculus of Construction

2.3.2 Calculus of Inductive Construction

Flesh out the details of Calculus of construction and Inductive construction

Write here about syntax and semantics of CIC

2.3.3 Dependent Types

Type system is a wide spectrum ranging from Scheme where type is runtime concept to Haskell to Coq

2.3.3.1 Example : Dependent Type Lambda Calculus

Lambda Calculus encoded in Coq using Inductive data type.

2.3.3.2 Correct by Construction

Combing program with proofs leads to one stop solution, mainly correct by construction

Well typed program can't go wrong. Give a example of dependent type lambda calculus(Dirk's white board) Hello World is dependent type Vector

2.3.3.3 Type vs. Prop : Code Extraction

explain here the difference between Prop and Types.

It's good starting point to tell the reader that we have two definitions, one in type and other in prop. Why ? Because Type computes, but it's not very intuitive for human inspection while Prop does not compute, but it's very intuitive for human inspection. We have connected that the definition expressed in Type is equivalent to Prop definition.

2.3.3.4 Gallina : The Specification Language

The example, dependent type lambda calculus, I gave in 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. It's not suitable for large proof development, and 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.

Can I give a simple example to demonstrate the difference between proof build directly in Gallina and proof build using tactics

2.3.4 Trusting Coq proofs

The fundamental question for trusting the Coq proofs is two fold: i) is the logic (CIC) sound ?, ii) is the implementation correct ?. The logic has already been reviewed by many peers and proved correct using some meta-logic. The Coq implementation itself can be partitioned into two parts: i) Validity Checker (Small kernel), ii) Tactic language to build the proofs. We lay our trust in validity checker, because it's small kernel. If there is bug in tactic language which often is the case then build proof would not pass the validity checker.

Try to write here how Fuzzer failed to find bugs in Compcert.

2.4 Summary

Paves the path to Cryptography.

2.5 Cryptography

Write some basic crypto stuff

2.5.1 Homomorphic Encryption

Add the details of homomorphic encryption

2.5.1.1 El-Gamal Encryption Scheme

Give both additive and multiplicative

2.5.1.2 Pallier Encryption Scheme

Write some description

2.5.2 Commitment Schemes

2.5.2.1 Hash Based Commitment Scheme

2.5.2.2 Discrete Logarithm Based Commitment Scheme

Pedersen's Commitment Scheme

2.5.3 Zero Knowledge Proof

Details from

2.5.4 Sigma Protocol : Efficient Zero Knowledge Proof

2.6 Summary

Chapter 3

Schulze Method : Evidence Carrying Computation

The correctness of vote counting in electronic voting is one of the main pillars that engenders trust in electronic voting. Even though, assuming that election results are verifiable, a bug in counting software which can cause a incorrect result and only caught later during verification phase would create a atmosphere of distrust among voters about electronic voting. The present state of art in vote counting leaves much to be desired: while some jurisdictions publish the source code of vote counting for public scrutiny, others treat the code as commercial in confidence. None of the systems used today in real world elections are formally verified. In this chapter, we formally specify the Schulze method, and the corner stone of our formalization is a correct by construction dependent inductive data type that represents all correct executions. Every inhabitant of this type not only produces a final result, but also all the intermediate steps that lead to result. Using these intermediate steps, any external verifier can verify the final result. As a consequence, we do not even need trust the execution of the (verified) algorithm: the correctness of a particular run of the vote counting code can be verified on the basis of the evidence for correctness that is produced along with determination of election winners.

Chapter overview: In this chapter, we explain the Schulze algorithm, followed by formal specification and

3.1 Schulze Algorithm

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 m candidates $C = \{c1, \ldots, cm\}$, 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 $b(c) > b(d)$.

- 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 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.

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

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

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

- The winning set (always non empty) is defined as

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

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 $k \in Z$ 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's a path p from c to d which 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.

3.1.1 An Example

Todo : Add a example with picture Draw a image here using TikZ which shows the Condercet paradox and resolve by the Schulze Method. Take the example from PhD pre

3.2 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.

```
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 <> nil.
```

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 $k \in Z$. The notion of winning candidate is that it beats every other candidates, 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 for human to inspect the definitions, and ascertain the correctness of formalization. As we discussed in **Type vs. Prop** section of first chapter, the main reason for having 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, 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 the declaring some one 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*, i.e. the produced result is correct, and convincing to all, specially 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 certification*: 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 $k \in Z$ and a margin function $m : C \times C \to Z$, 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 $V_k : Pow(C \times C) \to Pow(C \times C)$, defined by

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

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

$$W_k(R) = \{(c, e) \in C^2 \mid m(c, e) < k \text{ and } (m(c, d) < k \text{ or } (d, e) \in R \text{ for all } d \in C)\}$$

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 $k \in Z$. 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 : forall c d, marg c d >= k -> PathT k c d
| consT : forall c d e, marg c d >= k -> PathT k d e -> PathT k c e.
```

The only difference between type level paths (of type PathT) and (propositional) paths defined earlier is the fact that the former are proper types, not propositions, and are therefore not erased during extraction. Given the above, we have the following type-level definitions of winning (and dually, non-winning) for Schulze counting:

```
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.
```

The main result of this section is that type level and propositional evidence for winning (and dually, not winning) a Schulze election 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 biimplication only on propositions.

The proof of the first statement is completely straight forward, as the type carries all the information needed to establish the propositional winning condition. For the second statement above, we introduce 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 Carr'e [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 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 generalized margin. We show that candidate *c* is losing then there is a candidate *d* and generalized margin between candidate *d* and *c* is more that generalized 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.
```

Todo : Write a notation Chapter. What does it mean to be exists, and existsT

The proof of lemma *iterated_marg_loses_type* is not straight forward because Coq does not allow the case analysis on `Prop` when the goal in not in `Prop`, hence, we can not eliminate `exists n, P n` in order to show `existsT n, P n`. The idea is basically turning proposition level existence, *exists*, to type level existence, *existsT*. We can 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 co-closed 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, 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,

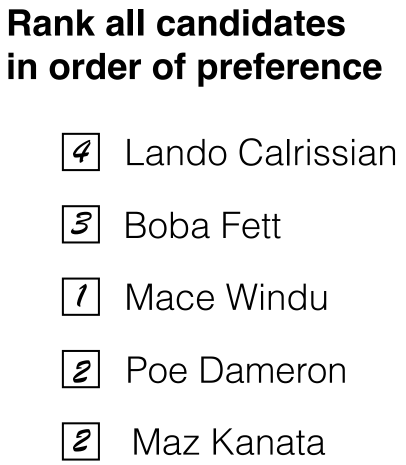
- an integer k and a path of strength $\geq k$ from the winning candidate to the other candidate

- a co-closed set that witnesses that no path of strength $> k$ exists in the opposite direction.

It is precisely this additional data (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 3.5, 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.

3.2.1 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.



Rank all candidates in order of preference

4	Lando Calrissian
3	Boba Fett
1	Mace Windu
2	Poe Dameron
2	Maz Kanata

In practice, ballots are implemented by asking voters to put numerical preferences against the names of candidates as represented by the image on the right. The most natural representation of a ballot is therefore a function $b : C \to Nat$ 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 $b : C \to 1 + Nat$ (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 stages 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 n`, for `n` of type `State`, represents a correct execution of the voting protocol up to reaching state `n`. 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) ->
  Count bs (partial (us, []) m)         (* zero margin      *)
| cvalid u us m nm inbs : Count bs (partial (u :: us, inbs) m) ->
  (forall c, (u c > 0)%nat) ->          (* u is valid       *)
  (forall c d : cand,
    ((u c < u d) -> nm c d = m c d + 1) (* c preferred to d *) /\
    ((u c = u d) -> nm c d = m c d)     (* c, d rank equal  *) /\
    ((u c > u d) -> nm c d = m c d - 1))(* d preferred to c *) ->
  Count bs (partial (us, inbs) nm)
| cinvalid u us m inbs : Count bs (partial (u :: us, inbs) m) ->
  (exists c, (u c = 0)%nat)             (* u is invalid     *) ->
  Count bs (partial (us, u :: inbs) m)
| fin m inbs w (d : (forall c, (wins_type m c) + (loses_type m c))):
  Count bs (partial ([], inbs) m)       (* no ballots left  *) ->
  (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 ballot 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.

3.2.2 All Schulze Election 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. In Coq, we use a type-level existential quantifier `existsT` where `existsT (x:A), P` stands for $\Sigma_{\texttt{x:A}}\texttt{P}$.

```
  Theorem schulze_winners: forall (bs : list ballot),
    existsT (f : cand -> bool) (p : Count bs (winners f)), 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 3.2. 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.

3.3 Scrutiny Sheet and Experimental Result

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 is can then be understood as a state of vote counting where election winners have been determined. Our main theorem 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*.

Todo : This already introduces software independence. Try to refer to that section. Universal verifiability.

From the perspective of electronic voting, this means that we no longer need to trust the hardware and software that was employed to obtain the election result, as the generated certificate can be verified independently. In the literature on electronic voting, this is known as *verifiability* and has been recognised as one of the cornerstones for building trust in election outcomes Chaum [2004], and is the only answer to key questions such as the possibility of hardware malfunctions, or indeed running the very software that has been claimed to count votes correctly.

The certificate that is produced by each run of our extracted Schulze vote tallying algorithm consists of two parts. The first part details the individual steps of constructing the margin function, based on the set of all ballots cast. The second part presents evidence for the determination of winners, based on generalised margins. For the construction of the margin function, every ballot is processed in turn, with the margin between each pair of votes updated accordingly. The heart of our work lies in this second part of the certificate. To demonstrate that candidate c is an election winner, we have to demonstrate that the generalised margin between c and every other candidate d is at least as large than the generalised margin between d and c. Given that the generalised margin between two candidates c and d is determined in terms of paths $c, c_1, \dots, c_n, d$ that join c and d, we need to exhibit

- evidence for the existence of a path p from c to d
- evidence for the fact that *no* path q from d to c is stronger than p

where the strength of a path $p = (c_0, \dots, c_{n+1})$ is the minimum $\min\{m(c_i, c_{i+1}) \mid 0 \leq i \leq n\}$ of the margins between adjacent nodes. While evidently a path itself is evidence for its existence, the *non-existence* of paths with certain properties is more difficult to establish. For this, we uses a coinductive approach. As existence of a path with a given strength between two candidates can be easily phrased as an inductive definition, the *complement* of this predicate arises as a greatest fixpoint, or equivalently as a coinductively defined predicate (see e.g. Kozen and Silva [2016]). This allows us to witness the non-existence of paths by exhibiting co-closed sets.

Coq's extraction mechanism then allows us to turn this 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 co-closed 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 3.2.

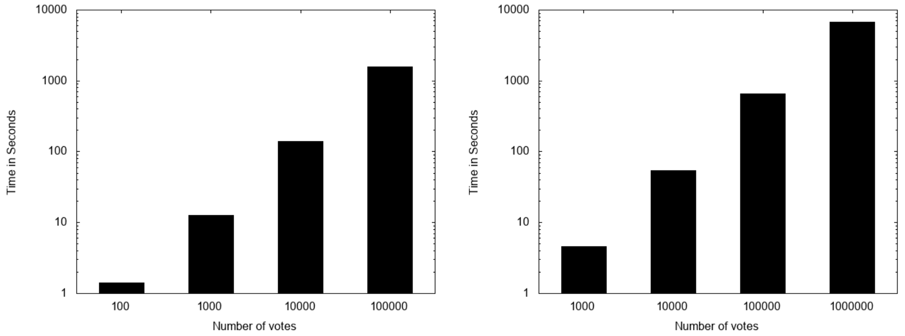
```
V: [A3 B1 C2 D4,..], I: [], M: [AB:0 AC:0 AD:0 BC:0 BD:0 CD:0]
-----------------------------------------------------------
```

```
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 strenght 3, 4-coclosed set:
    [(B,A),(C,A),(C,B),(D,A),(D,B),(D,C)]
  for C: path A --> C of strenght 3, 4-coclosed set:
    [(B,A),(C,A),(C,B),(D,A),(D,B),(D,C)]
  for D: path A --> D of strenght 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



(a) Direct Extraction

(b) Extraction using Haskell Integers

Figure 3.1: Experimental Results

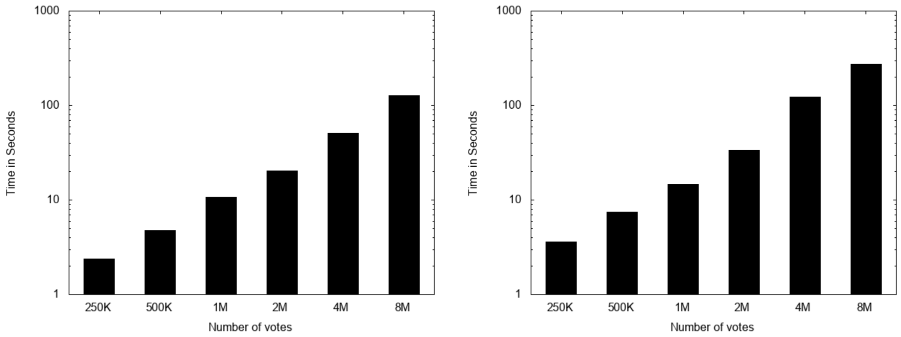
sheet provides an *orthogonal* perspective on the data and how it was used to determine the election result.

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. The results are displayed in Figure 3.2(a) using a logarithmic scale.

3.4 Scaling it count millions of Ballots

We investigated 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. Below, we include our timing graphs, based on randomly generated ballots while keeping number of candidates constant i.e. 4.
On the left, we report timings (in seconds) for the computation of winners, whereas on the right, 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 no of ballots and cubic in 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 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.



(a) Computation of Winners

(b) Computation of Winners and Certificate

Figure 3.2: Experimental Results

3.5 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. In this paper, 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 are many freely available implementations, 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 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 spotcheck 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 co-closed – 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 election officials and 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 3.2) 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 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.

3.6 Summary

Our formalization achieves *Correctness*, *Practicality*, and (universal) *Verifiability*. The major wrinkle 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 pros and cons of this formalization:

- Pros
  - 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 because of the generated certificates.
  - Verifiability: 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.
- Cons
  - Privacy : There is no privacy, and possibly susceptible to coercion, because the ballots involved are simply plaintext.

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

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.

Chapter 4

Homomorphic Schulze Algorithm : Axiomatic Approach

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 will try to achieve both: privacy, using encryption, and (universal) verifiability, using zero knowledge proof. The encryption of ballots is crucial to maintaining integrity and anonymity in electronic voting schemes. It enables, amongst other things, each voter to verify that their encrypted ballot has been recorded as cast, by checking their ballot against a bulletin board.

Chapter Outline: Turn this into chapter outline once you finish everything. We present a verifiable homomorphic tallying scheme for the Schulze method that allows verification of the correctness of the count—on the basis of encrypted ballots—that only reveals the final tally. We achieve verifiability by using zero knowledge proofs for ballot validity and honest decryption of the final tally. Our formalisation takes places inside the Coq theorem prover and is based on an axiomatisation of cryptographic primitives, and our main result is the correctness of homomorphic tallying. We then instantiate these primitives using an external library and show the feasibility of our approach by means of case studies.

4.1 Introduction

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 substantiated with verifiable evidence of their correctness while retaining the secrecy of the individual ballot Bernhard et al. [2017]. Technically, the notion of "verifiable evidence" is captured by the term *end-to-end (E2E) verifiability*, that is

- every voter can verify that their ballot was cast as intended
- every voter can verify that their ballot was collected as cast
- everyone can verify final result on the basis of the collected ballots.

While end-to-end verifiability addresses the basic assumption that no entity (software, hardware and participants) are inherently trustworthy, ballot secrecy addresses the privacy problem. Unfortunately, it appears as if coercion resistance is not achievable in the remote setting without relying on overly optimistic—to say the least—assumptions. A weaker property called receipt-freeness captures the idea that an honest voter—while able to verify that their ballot was counted—is required to keep no information that a possible coercer could use to verify how that voter had voted.

End to end verifiability and the related notation of software independence Rivest [2008] have been claimed properties for many voting schemes. Küsters, Truderung and Vogt Küsters et al. [2010] gave a cryptographic formulation whose value is highlighted by the attacks it revealed against established voting schemes Küsters et al. [2012].

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]. (Separate techniques exist to prevent ballot box stuffing and to guarantee cast-as-intended.) 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 ZKP 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 proofs.

This chapter addresses the problem of verifiable homomorphic tallying for a preferential voting scheme, the Schulze Method. We show how it can be implemented in a theorem prover to guarantee both provably correct and verifiable counting on the basis of encrypted ballots, relative to an axiomatisation of the cryptographic primitives. We then obtain, via program extraction, a provably correct implementation of vote counting, that we turn into executable code by providing implementations of the primitives based on a standard cryptographic library. We conclude by presenting experimental results, and discuss trust the trust base, security and privacy as well as the applicability of our work to real-world scenarios.

The Schulze Method. We give a brief overview of Schulze method for convenience of the reader. The Schulze Method Schulze [2011] is a preferential, single-winner vote counting scheme that is gaining popularity due to its relative simplicity while retaining near optimal fairness Rivest and Shen [2010]. A *ballot* is a rank-ordered list of candidates where different candidates may be given the same rank. The protocol proceeds in two steps, and first computes the *margin matrix* m, where $m(x,y)$ is the relative margin of x over y, that is, the number of voters that prefer x over y, minus the number of voters that prefer y over x. In symbols, given a collection B of ballots,

$$m(x,y) = \sharp\{b \in B \mid x <_b y\} - \sharp\{b \in B \mid y <_b x\}$$

where $\sharp$ denotes cardinality, and $<_b$ is the preference relation encoded by ballot b. We note that $m(x,y) = -m(y,x)$, i.e. the margin matrix is symmetric. In a second step, a *generalised margin* g is computed as the strongest path between two candidates

$$g(x,y) = max\{str(p) \mid p \text{ path from } x \text{ to } y\}$$

where a path from x to y is simply a sequence $x = x_0, \ldots, x_n = y$ of candidates, and the strength

$$str(x_0, \ldots, x_n) = min\{m(x_i, x_{i+1}) \mid 0 \leq i < n\}$$

is the lowest margin encountered on a path. Informally, one may think of the generalised margin $g(x,y)$ as transitive accumulated support for x over y. We say that x beats y if $g(x,y) \geq g(y,x)$ and a winner is a candidate that cannot be beaten by anyone. That is, w is a *winner* if $g(w,x) \geq g(x,w)$ for all other candidates x. Note that winners may not be uniquely determined (e.g. in the case where no ballots have been cast).

In previous work Pattinson and Tiwari [2017] we have demonstrated how to achieve verifiability of counting plaintext ballots by producing a verifiable *certificate* of the count, where ballot privacy and receipt freeness are not addressed. The certificate has two parts: The first part witnesses the computation of the margin matrix where each line of the certificate amounts to updating the margin matrix by a single ballot. The second part witnesses the determination of winners based on the margin matrix. In the first phase, i.e. the computation of the margin matrix, we perform the following operations for every ballot:

1. if the ballot is informal it will be discarded

2. if the ballot is formal, the margin matrix will be updated

The certificate then contains one line for each ballot and thus allows to independently verify the computation of the margin matrix. Based on the final margin matrix, the second part of the certificate presents verifiable evidence for the computation of winners. Specifically, if a candidate w is a winner, it includes:

1. an integer k and a path of strength k from w to any other candidate

2. evidence, in the form of a co-closed set, of the fact that there cannot be a path of strength $> k$ from any other candidate to w.

Crucially, the evidence of w winning the election *only* depends on the margin matrix. We refer to Pattinson and Tiwari [2017] for details of the second part of the certificate as this will remain unchanged in the work we are reporting here.

4.2 Verifiable Homomorphic Tallying

The realisation of verifiable homomorphic tallying that we are about to describe follows the same two phases as the protocol: We first homomorphically compute the margin matrix, and then compute winners on the basis of the (decrypted) margin. The computation also produces a verifiable certificate that leaks no information about individual ballots other than the (final) margin matrix, which in turn leaks no information about individual ballots if the number of voters is large enough. As for counting of plaintext ballots, we disregard informal ballots in the computation of the margin. In accord with the two phases of computation, the certificate consists of two parts: the first part evidences the correct (homomorphic) computation of the margin, and the second part the correct determination of winners. We describe both in detail.

Format of Ballots. 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 $b : C \rightarrow Nat$ 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 $b : C \rightarrow Nat$ over a (finite) set C of candidates.

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.

We instead represent ballots as matrices $b(x,y)$ where $b(x,y) = +1$ if x is preferred over y, $b(x,y) = -1$ if y is preferred over x and $b(x,y) = 0$ if x and y are equally preferred.

While the advantage of the first representation is that each ranking function is necessarily a valid ranking, the advantage of the matrix representation is that the computation of the margin matrix is simple, that is

$$m(c,d) = \sum_{b \in B} b(x,y)$$

where B is the multi-set of ballots (in matrix form), and can moreover be transferred to the encrypted setting in a straight forward way: if ballots are matrices $e(x,y)$ where $e(x,y)$ is the encryption of an integer in $\{-1, 0, 1\}$, then

$$encm = \bigoplus_{encb \in EncB} encb(x,y) \tag{4.1}$$

where $\oplus$ denotes homomorphic addition, $encb$ is an encrypted ballot in matrix form (i.e. decrypting $encb(x,y)$ indicates whether x is preferred over y), and $EncB$ is the multi-set of encrypted ballots. The disadvantage is that we need to verify that a matrix ballot is indeed valid, that is

- that the decryption of $encb(x,y)$ is indeed one of $1, 0$ or -1
- that $encb$ indeed corresponds to a ranking function.

Indeed, to achieve verifiability, we not only need *verify* that a ballot is valid, we also need to *evidence* its validity (or otherwise) in the certificate.

Validity of Ballots. By a plaintext (matrix) ballot we simply mean a function $b : C \times C \to Z$, where C is the (finite) set of candidates. A plaintext ballot $b(x,y)$ is *valid* if it is induced by a ranking function, i.e. there exists a function $f : C \to Nat$ such that $b(x,y) = 1$ if $f(x) < f(y)$, $b(x,y) = 0$ if $f(x) = f(y)$ and $b(x,y) = -1$ if $f(x) > f(y)$. A *ciphertext (matrix) ballot* is a function $encb : C \times C \to CT$ (where CT is a chosen set of ciphertexts), and it is valid if its decryption, i.e. the plaintext ballot $b(x,y) = dec(encb(x,y))$ is valid (where dec denotes decryption).

For a plaintext ballot, it is easy to decide whether it is valid (and should be counted) or not (and should be discarded). We use shuffles (ballot permutations) to evidence the validity of encrypted ballots. One observes that a matrix ballot is valid if and only if it is valid after permuting both rows and columns with the same permutation. That is, $b(x,y)$ is valid if and only if $b'(x,y)$ is valid, where

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

and $\pi : C \to C$ is a permutation of candidates. (Indeed, if f is a ranking function for b, then $f \circ \pi$ is a ranking function for b'). As a consequence, we can evidence the validity of a ciphertext ballot $encb$ by

- publishing a shuffled version $encb'$ of $encb$, that is shuffled by a secret permutation, together with evidence that $encb'$ is indeed a shuffle of $encb$
- publishing the decryption b' of $encb'$ together with evidence that b' is indeed the decryption of $encb'$.

We use zero-knowledge proofs 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 $encm$, and for each ciphertext ballot $encb$ contains

1. a shuffle of $encb$ together with a ZKP of correctness
2. decryption of the shuffle, together with a ZKP of correctness
3. the updated margin matrix, if the decrypted ballot was valid, and
4. the unchanged margin matrix, if the decrypted ballot is not valid.

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

5. the fully constructed margin, together with its decryption and ZKP of honest decryption after counting all the ballots

6. publishes the winner(s), together with evidence to substantiate the claim

Cryptographic primitives. We require an additively homomorphic cryptosystem to compute the (encrypted) margin matrix according to Equation 4.1 (this implements Item 3 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 1 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 a 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

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 completness, as it is identicial to the existing work on plaintext counting Pattinson and Tiwari [2017]. For each of the winners w and each candidate x we publish

- a natural number $k(w, x)$ and a path $w = x_0, \ldots, x_n = x$ of strength k
- 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 $\geq k$. We refer to *op.cit.* for the (formal) proofs of the fact that existence of co-closed sets witnesses the winning conditions.

4.3 Realisation in a Theorem Prover

We formalise homomorphic tallying for the Schulze Method inside the Coq theorem prover Bertot et al. [2004]. Apart from supporting an expressive logic and (crucial for us) dependent inductive types, Coq has a well developed extraction facility that we use to extract proofs into OCaml programs. Indeed, our basic approach is to first formally define the notion of a valid certificate, and then prove that a valid certificate can be obtained from any set of (encrypted) ballots. Extracting this proof as a programme, we obtain an executable that is correct by construction.

The purpose of this work was 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 El Gamal Gamal [1984]).

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 difference between the keyword `Variable` and `Axiom` is purely syntactic, and in our case, used as a convenience for extraction. In the above, the first two functions, `decrypt_message` and `construct_zero_knowledge_decryption_proof` are *generation* primitives, whereas `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 (dependent, inductive) 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 dependent inductive 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.

Inductive type *ECount* with all the required constructors filled with *state data*, *verification data*, and *correctness constraint*.

```
 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)
  | 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) ->
        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)
  | 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) ->
        ~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)
  | 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)
  | 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).
```

Todo : Dirk: Should I add more details ?
The first constructor, Ecax kick-starts the count, and ensures that

- all ballots are initially uncounted

- margin matrix is an encryption of the zero matrix

The first constructor, as well as all the others, require

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).
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.

We elide the description of the third constructor that is applied when an invalid ballot is being encountered (the only difference is that the margin matrix is not being updated). Counting finishes when there are no more uncounted ballots, 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.

The last constructor 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 identical to our previous formalisation of the Schulze Method (for plaintext ballots), and we refer to Pattinson and Tiwari [2017] for more details.

4.4 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 – give 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 first 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. Plaintext counting has been formalised, and verified, in the precursor paper Pattinson and Tiwari [2017]. 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 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` 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.

4.5 Extraction and Experiments

As already mentioned, we are using the Coq extraction mechanismLetouzey [2003] to extract programs from existence proofs[1]. In particular, we extract the proof of the Theorem `pschulze_winners`, given in Section 4.4 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 4.3, the formal development does not presuppose any specific implementation of the cryptographic primitives, and we assume the existence of cryptographic infrastructure. From the perspective 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.

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

To fill this hole, we implement the cryptoraphic primitives with help of the UniCrypt libraryLocher and Haenni [2014]. UniCrypt is a freely available library, written in Java, that provides nearly all of the required functionality, 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. Due to differences in the type structure between Java and OCaml, mainly in the context of sub-typing, this was done in the form of an OCaml wrapper around Java data structures. 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 is 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>
```

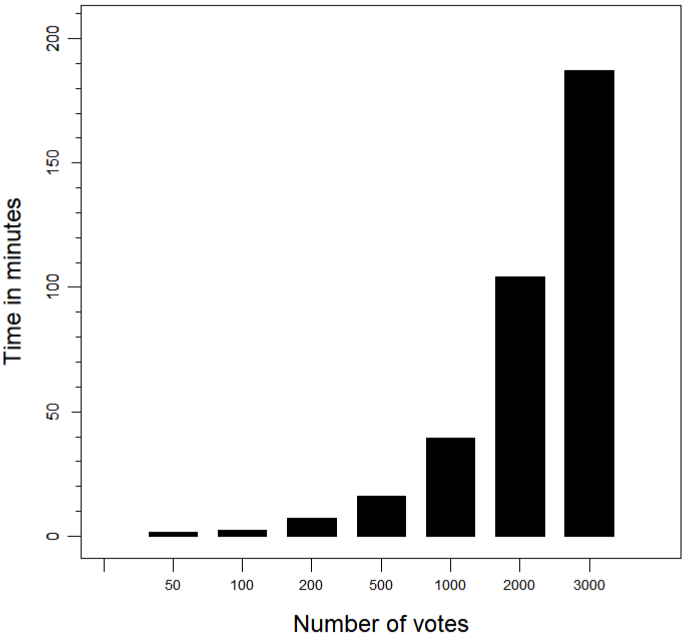
where the notion of evidence for winning and losing is as in the plaintext version of the protocol Pattinson and Tiwari [2017]. A glimpse of a concrete certificate for an election, and, in other words, it is a trace of the computation which can be used as a checkable record to verify the outcome of election. For the space consideration, 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:
       [(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 8GB of RAM for three candidates and randomly generated ballots. 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.



Personal Experience: This project turned out to be very challenging than I anticipated when I started it. Dirk : Would it be a good idea to write my personal experience about this project ? It was very challenging from proving the lemmas to writing the OCaml binding. I would like to write a section on it, but, only if, it is appropriate

4.6 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 expertiments 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.

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 Pattinson and Tiwari [2017]. Given that both parts are practically efficient by themselves, scalability is merely the question of engineering a more efficient coupling.

In a nutshell, the pros and cons of this formalization:

- Pros
  - 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 himself.
  - Verifiability: The outcome of any election can be verified by any third party because of the generated certificates. However, certificates are only accessible to someone having specialized knowledge of cryptography.
- Cons
  - Verifiability: The nature of certificates in this case is very complex, and they can only be scrutinize by someone having specialized knowledge of cryptography which decreases the pool of potential scrutinizers dramatically.
  - Practicality: The extracted code hardly counts 10,000 ballots in 25 hours which is certainly not the case in real life election.
  - Correctness: We used external unverified libraries for cryptographic code. These libraries could be buggy and may produce a wrong result. This, per say, is not a problem because it will be caught during the certificate checking by any independent party, but, it may create a atmosphere of distrust among voters about electronic voting.

Our formalization leaves some gaps which needed to be filled:

- A formally verified cryptographic library to fill the *Practicality* gap
- A formally verified checker to easy the auditing of election to fill the *Scrutineers* gap

Todo : Rewrite this once you finish the formally verified checker Developing a formally verified library to fill the practicality gap would probably be itself a PhD, so we chose the other path. Technically, we have not developed the certificate checker for our scheme, but for IACR election which is simpler than ours.

In the next chapter, we will focus on bits and pieces of formally verified certificate checker. The reason for formally verifying the certificate checker is *correctness*. Recall that the crucial requirement of certificate is: i) correctness, and ii) simplicity. Also, to affirm the public trust in electronic voting, it is a good idea to provide a reference checker and make it open source. Even though, the reference checker is formally verified, the idea behind making it open source is to gain the public trust via scrutiny or openness. Providing a reference checker, also, opens the gate for debate in case of someone's implementation for checking certificate diverges from the reference checker. In the case of diverging situation, there are two possibility, either the reference solution is verified using wrong assumption, or the implementation itself is wrong. The first situation is certainly not very pleasant because it would deteriorate the public trust, but nonetheless, it is always good to have openness in democracy to make it more strong.

Chapter 5

Scrutiny Sheet : Software Independence (or Universal Verifiability)

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

Roald Dahl

Getting everything right in electronic voting is very difficult, and assuming, for a moment, that everything is correct, convincing this fact to every stakeholder and any independent third party is almost next to impossible. As we stated in earlier chapter [Give a link to the section] that software is complex artefact and, often, poorly design and tested. It should not come as a surprise to anyone that we are far more competent in producing the incorrect software than producing a provable correct one. In order to tackle the software complexity problem and ensure the public trust in process, Ronald Rivest and John Wack proposed the term "software-independence": [1]

A voting system is software independent if an undetected change or error in its software can not cause an undetected change or error in an election outcome.

Software-independence is weaker notion than end-to-end verifiability. Software independence put forward the idea of detection (and possible correction) of outcome of election due to software bug, while the end-to-end verifiability makes the whole process transparent without trusting any component involved in the process (including any hardware and software) [Bernhard et al., 2017]. Recall that end-to-end verifiability:

- 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 final result on the basis of the collected ballots.

[1] https://people.csail.mit.edu/rivest/RivestWack-OnTheNotionOfSoftwareIndependenceInVotingSystems.pdf

Benaloh [2006] has given a detailed overview about achieving each step of end-to-end verifiability, we are only concern about the last phase, i.e. `Tallied-as-cast`. Scrutiny sheet not only provides the `Tallied-as-cast` (also known as universal verifiability) assuming that first two, `Cast-as-intended` and `Collected-as-cast`, hold, but it makes our voting system software independent. It is worth noting that any bug or malicious behaviour in software used to produce the result can not go undetected if the results produced were incorrect. The rationale is that any independent third party auditing/verifying the result would write his own checker to accomplish the task, and statistically, if more people are auditing the election, then it is highly unlikely that incorrect result produced by buggy/malicious software would survive for long.

Chapter Overview[Possibly rewriting after finishing the chapter] In the closing remark of last chapter, we argued that it is always a good idea to provide a open source reference checker. This chapter focuses on the technical details needed to develop a certified checker for election conducted on encrypted ballots. In section [refer the section], we discuss the structure of encrypted-ballot scrutiny sheet, elaborate the relevant details to understand the certificate, section [some number] discusses about what it means to verify the zero-knowlege-proof of different statement. Developing a formally verified certificate checker for our certificate could have taken long time, so in order to demonstrate the idea we have the taken the IACR 2018 election which we discuss in section [some number].

1. first paste the certificate
2. take each piece of information, and show that how can it be verified
3. Flesh the details of honest decryption zero knowledge proof
4. Flesh the details of permutation shuffle

5.1 Certificate : Ingredient for Verification

There are two notion of verification. Software verification and election verification. A electronic voting scheme implemented in Coq is verified implementation, but it does not imply that method itself is veri-

5.1.1 Plaintext Ballot Certificate

Flesh out the details needed here for writing proof checker

5.1.2 Encrypted Ballot Certificate

Flesh out the details needed here for writing proof checker

5.2 Proof Checker

Write the details of proof checker for both certificates

5.2.1 A Verified Proof Checker : IACR 2018

Write some details about proof checker.

5.3 Summary

Write some advantage of proof checker for certificates. To create the mass scrutineers, all we need is a simple proof checker which would take proof certificate as input and spit true or false. If it's true then we accept the outcome of election otherwise something wrong.

Chapter 6

Machine Checked Schulze Properties

Not planned but hopefully it will done

6.1 Properties

List some properties which it follows with pictures ?

6.1.1 Condercet Winner

6.1.2 Reversal Symmetry

6.1.3 Monotonicity

6.1.4 Schwartz set

Chapter 7

Conclusion

Same as the last chapter, introduce the motivation and the high-level picture to readers, and introduce the sections in this chapter.

7.1 Related Work

Here goes the details for related work

7.2 Future Work

Possibility of future work

7.2.1 Formalization of Cryptography

7.2.2 Integration with Web System : Helios

Bibliography

Aguillon, J. Ocaml ↔ Java Interface. https://github.com/Julow/ocaml-java. Accessed on April 29, 2019. (cited on page 55)

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 1)

Arkoudas, K. and Rinard, M. C., 2005. Deductive runtime certification. *Electr. Notes Theor. Comput. Sci.*, 113 (2005), 45–63. (cited on page 16)

Arrow, K. J., 1950. A difficulty in the concept of social welfare. *Journal of Political Economy*, 58, 4 (1950), 328–346. (cited on page 5)

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 May 14, 2017. (cited on page 15)

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 4)

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 42 and 46)

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 12)

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 42)

Benaloh, J., 2006. Simple verifiable elections. In *Proceedings of the USENIX/Accurate Electronic Voting Technology Workshop 2006 on Electronic Voting Technology Workshop*, EVT'06 (Vancouver, B.C., Canada, 2006), 5–5. USENIX Association, Berkeley, CA, USA. http://dl.acm.org/citation.cfm?id=1251003.1251008. (cited on page 62)

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 4)

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 page 6)

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 6, 41, and 61)

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 5 and 48)

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 4)

Carr'e, B. A., 1971. An algebra for network routing problems. *IMA Journal of Applied Mathematics*, 7, 3 (1971), 273. (cited on page 28)

Carrier, M. A., 2012. Vote counting, technology, and unintended consequences. *St Johns Law Review*, 79 (2012), 645–685. (cited on pages 14, 15, and 16)

Chaum, D., 2004. Secret-ballot receipts: True voter-verifiable elections. *IEEE Security & Privacy*, 2, 1 (2004), 38–47. (cited on page 34)

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 45)

Conway, A.; Blom, M.; Naish, L.; and Teague, V., 2017. An analysis of New South Wales electronic vote counting. In *Proc. ACSW 2017*, 24:1–24:5. (cited on pages 14 and 15)

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 4)

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 4)

DELAUNE, S.; KREMER, S.; AND RYAN, M., 2010a. *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 4)

DELAUNE, S.; KREMER, S.; AND RYAN, M., 2010b. 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 page 6)

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 page 4)

ELECTIONS ACT, 2016. Electronic voting and counting. avalilable via http://www.elections.act.gov.au/elections_and_voting/electronic_voting_and_counting, retrieved May 14, 2017. (cited on page 15)

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 1)

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 4)

GAMAL, T. E., 1984. A public key cryptosystem and a signature scheme based on discrete logarithms. In *CRYPTO*, vol. 196 of *Lecture Notes in Computer Science*, 10–18. Springer. (cited on page 48)

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 page 4)

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 5)

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 4)

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 42)

Goldwasser, S.; Micali, S.; and Rackoff, C., 1985. The knowledge complexity of interactive proof-systems (extended abstract). In *STOC*, 291–304. ACM. (cited on page 42)

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 1 and 12)

Helios, 2016. The helios voting system. Http://heliosvoting.org/, accessed June 25, 2016. (cited on page 4)

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 42)

Hood, C., 2001. Transparency. In *Encyclopedia of Democratic Thought* (Eds. P. B. Clarke and J. Foweraker), 700–704. Routledge. (cited on page 15)

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 11)

Kozen, D. and Silva, A., 2016. Practical coinduction. *Mathematical Structures in Computer Science*, (Feb. 2016), 1–21. (cited on page 34)

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 4)

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 page 39)

Küsters, R.; Truderung, T.; and Vogt, A., 2010. Accountability: definition and relationship to verifiability. In *ACM Conference on Computer and Communications Security*, 526–535. ACM. (cited on page 42)

Küsters, R.; Truderung, T.; and Vogt, A., 2012. Clash attacks on the verifiability of e-voting systems. In *IEEE Symposium on Security and Privacy*, 395–409. IEEE Computer Society. (cited on page 42)

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 page 6)

Landin, P. J., 1964. The mechanical evaluation of expressions. *The Computer Journal*, 6, 4 (1964), 308. (cited on page 36)

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 54)

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 page 55)

Meijer, A., 2014. Transparency. In *The Oxford Handbook of Public Accountability* (Eds. M. Bovens; R. E. Goodin; and T. Schillemans), 507–524. Oxford University Press. (cited on page 15)

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 13)

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 13)

O'Neill, O., 2002. *A Question of Trust*. Cambridge University Press. (cited on page 14)

Otten., J., 2003. Fuller disclosure than intended. (2003). www.votingmatters.org.uk. (cited on pages 4, 38, and 41)

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 page 4)

Pattinson, D. and Tiwari, M., 2017. Schulze voting as evidence carrying computation. In *Proc. ITP 2017*, vol. 10499 of *Lecture Notes in Computer Science*, 410–426. Springer. (cited on pages 43, 48, 52, 53, 56, and 59)

Pattinson, D. and Verity, F., 2016. Modular synthesis of provably correct vote counting programs. In *Proc. E-Vote-ID 2016*. To appear. (cited on page 4)

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 page 46)

Rivest, R. L., 2008. On the notion of Ôsoftware independenceÕin voting systems. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366, 1881 (2008), 3759–3767. (cited on page 42)

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 pages 23 and 42)

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 4)

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 5, 23, 30, and 42)

SCHÜRMANN, C., 2009. Electronic elections: Trust through engineering. In *Proc. RE-VOTE 2009*, 38–46. IEEE Computer Society. (cited on page 16)

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 26)

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 16)

TARSKI, A., 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5, 2 (1955), 285—309. (cited on page 27)

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 45)

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 4)

VOGL, F., 2012. *Waging War on Corruption: Inside the Movement Fighting the Abuse of Power*. Rowman & Littlefield. (cited on page 14)

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 46 and 47)

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 1 and 11)

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 13)