Formally Verified Electronic Voting Scheme : A Case Study

Mukesh Tiwari

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

November 2019

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

Mukesh Tiwari
25 November 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

A democracy can be described as a system where every eligible participant has equal right to express their opinion(s) on different matters. One of the most important example of expressing the opinion is holding elections to elect the leader of country. During the elections, every eligible participant expresses their opinion on a paper, also known as ballot, by ranking the participating candidate in according to their preference. Later, once the cast finishes, a candidate is elected from participating candidates by combing the all choices of participant. The paper ballot method works great, except it is very time consuming, expensive and error prone. Moreover, paper ballot elections are not environment friendly, and it produces a lot of waste in form of discarded papers. In order to solve the problems posed by paper ballot, many countries are adopting electronic voting. Electronic voting is getting popular in many countries, and the reason for its popularity is cost-effective, faster result, and high voter turn out. Undeniably, electronic voting has helped Australia to ease the logistic challenges of elections because of its massive land size and sparsely population and save millions of dollars. it has helped India, the second most populous country with 900 million eligible voter, to declare 2019 election with 67 percent voter turn out (roughly 600 million) in 2 days. Estonia, a labour shortage country, has saved thousands of man hours, 11,000 working days, by using electronic voting [Est].

Despite all these benefits, electronic voting is a nightmare because a minuscule possible of going anything wrong in software or hardware could lead to a disaster, possibly inverting the results [Lewis et al.], [Halderman and Teague, 2015], [Aranha et al., 2019], [Feldman et al., 2007]. The nature of (electronic) data and ease of its manipulability/misinterpretation bring electronic voting many problems, which are

not present in paper ballot election, that makes it perfectly susceptible to delivering wrong and unverifiable result [Wolchok et al., 2010]. For instance, if a software program used in electronic voting for reading the ballots has byte order bug, or even if it depends on some other software which has byte order bug, then the interpretation of ballot would completely be different from the what the voter had in mind. More often than not, these software programs are configured incorrectly and run at the top of (untrusted) operating system and hardware [Kohno et al., 2004]. Usually, operating systems have millions of lines code (Linux has 15 millions lines) which exposes a large attack surface and could be exploit, possibly by current government or foreign country, for illegal gain. The worst, these software and hardware used in electronic voting process are commercial in confidence and treated as a black-box, and, most often, their source code or design is not open for public scrutiny [Australian Electoral Commission, 2013]. In addition, these software program take a pile of ballot and produce the result without producing any evidence about the correctness of result. As a consequence, from casting the ballot electronically and declaring winner based on cast (electronic) ballot, the whole process lacks, basic assumption of democracy, genuineness.

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

- Correctness: The produced results are correct, and convincing to all leaving no ground for suspicion.
- Coercion-resistance: A voter cannot cooperate with a coercer to prove anything about her choices.
- Eligibility: Only eligible voter can cast the ballot.
- 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.

In this thesis, we focus on privacy, correctness, and tallied-as-cast, the third part of end-to-end verifiability, property of an election. Furthermore, we assume that the

first two properties of end-to-end verifiability, cast-as-intended and collected as cast, hold for an election, i.e. the front end voting software interacting with voters is not changing the options of a voter, and every voter has verified that his ballot appears correctly on bulletin board.

1.2 Research Motivation and Contribution

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

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 answer these question by taking the Schulze method [Schulze, 2011] as an example and Coq [Bertot et al., 2004] theorem prover for implementing and proving the correctness of Schulze method. Even though Schulze method is not used in any democratic election to public office, the reason we went ahead with it because it has many interesting properties and, at the same time, it is non-trivial. Schulze's method elects a single winner based on preferential votes. 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. We will discuss more about the Schulze method in chapter 4, and its properties in chapter 7. The reason for choosing Coq is that it supports an expressive logic and dependent inductive types which is very crucial for us. We will discuss more about the Coq in chapter 3.

We demonstrate that it is possible to achieve correctness, privacy and (tallied-as-cast) verifiability in electronic voting. We achieve:

- `Correctness` by formally specifying the Schulze method inside Coq theorem prover, and prove the correctness properties. 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.

- `Privacy` by encryption. We use homomorphic-encryption to compute the finally tally without decrypting any individual ballot.

- `Verifiability` by tabulating the relevant data of election. We call it scrutiny-sheet/certificate. Achieving verifiability in a plain-text ballot counting is fairly straight forward, but it is not the same with encrypted ballot counting. To achieve verifiability in encrypted ballot counting, we augment the scrutiny sheet with zero-knowledge-proof for the each claim we make during the counting which can later be checked by any auditor.

In addition to these, we have also developed a formally verified certificate checker as a proof of concept for automating the auditing an election. Having said that, our certificate is very complex, and formalizing all the primitives involved would be fairly time consuming, so we have developed a proof of concept formally verified certificate checker for IACR 2018 (International Association of Cryptographic Research) election, relatively simple than ours, scrutiny sheet (chapter 6). Also, we have proved couple of properties, Condercet winner, and Reversal symmetry property of the Schulze method inside Coq theorem prover (chapter 7).

1.3 Cryptographic Blackbox

Our primary goal was not to verify the cryptographic primitives, but use them as a facilitator to achieve privacy (using encryption) and verifiability (using zero knowledge proof) in electronic voting. To achieve this goal, we have taken the axiomatic approach, and assumed the existence of cryptographic primitives inside Coq with the axioms about their correctness behaviour. These primitives, in general, provide functionality of encrypting a plain-text, decrypting a cipher-text, constructing a zero-knowledge-proof, and verifying a zero-knowledge-proof. Later, in extracted OCaml code, these functions are instantiated with Unicrypt[Locher and Haenni, 2014] function. We will discuss more on this in chapter 5. [1]

1.4 Publication

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

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.

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

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. Verifiable Homomorphic Tallying for the Schulze Vote Counting Scheme (VSSTE paper)

5. Verified Verifiers for Verifying Elections (CCS paper)

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

1.5 Related Work

There is an extensive work which addresses the different issues related of electronic voting protocols in symbolic model, but there are very few, to the best of my knowledge, that have used theorem provers to implement the voting protocol (counting algorithm) and verify its correctness properties. [Kremer and Ryan, 2005], and [Delaune et al., 2010b] have used pi-calculus to model and analyse various properties, such as 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 vote counting.

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].

1.6 Outline of the Chapters

[Rewrite when you are done with this thesis] Chapter 2 provides an overview of electronic voting around the world, problems in general, and rationale for formal verification of election voting software. Chapter 3 provides the overview of concept of Coq theorem prover and cryptographic primitives. Chapter 4 describes Schulze method, its formal specification, proof of correctness, experimental result, and scrutiny sheet. Chapter 5 describes verifiable homomorphic tally for Schulze method, its realization in theorem prover, experimental result, instructions to audit the scrutiny sheet. Chapter 6 focuses on the notion of software independence, and formalization of Sigma protocol. Chapter 7 (ongoing work) describes the properties of Schulze method. [Todo : Rewrite the last line when you have last chapter. Hopefully, that would be last day of writing :)] Finally, chapter 8 concludes the thesis, and some possible direction of future work.

1.7 Trivia

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

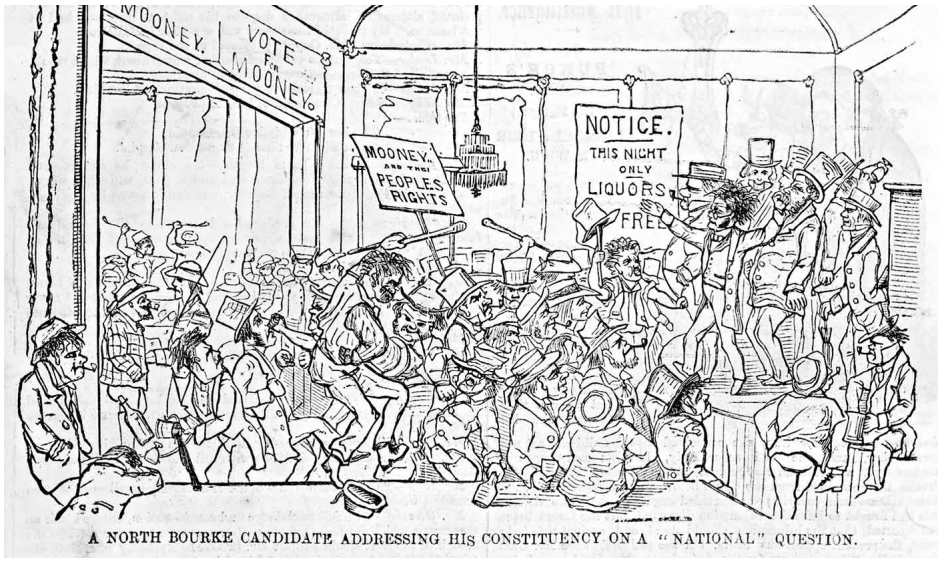


Figure 1.1: Election held in 1855 in Victoria, Australia was conducted in pub!

Chapter 2

Background

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

Counting ballots by hand is tedious, error prone, slow, and costly process. For example, the Senate election conducted in Western Australia in September 2013 was declared void by the high court because of the loss of 1370 votes. It was re-conducted in April 2014 at the cost of 20 Million AUD with additional delay in results [Aus]. Before introduction to electronic voting machines in India, it will take month to declare the result. As a consequence, many countries are now adopting electronic voting to alleviate the problems introduced by hand counting.

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

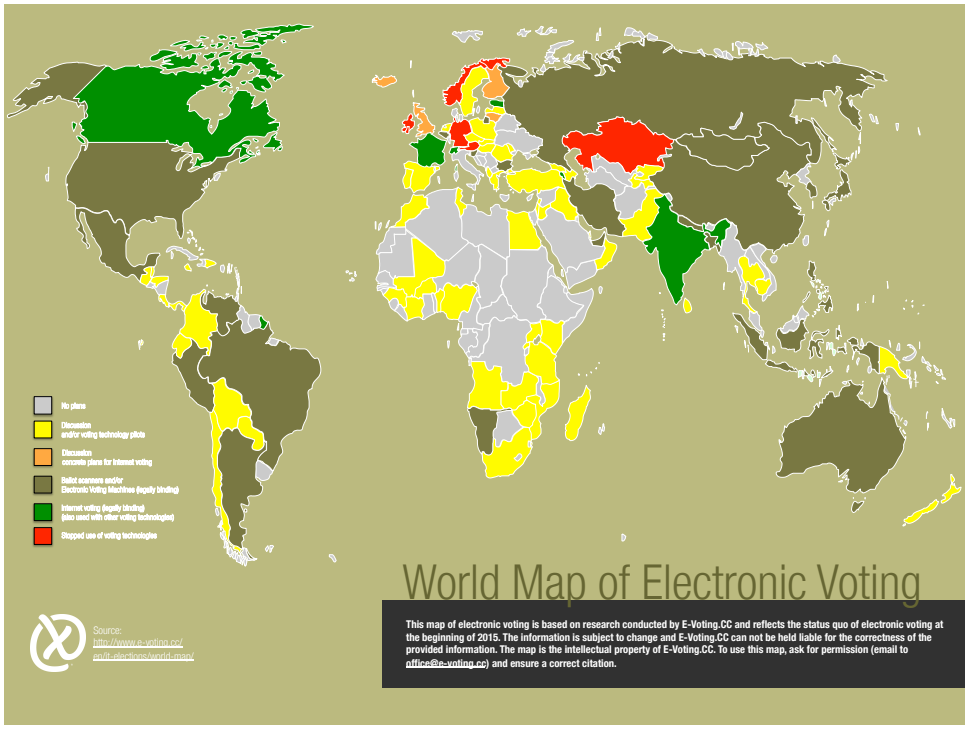


Figure 2.1: World map of Electronic Voting

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

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

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

> The legislature is not prevented from using electronic voting machines in the elections if the constitutionally required possibility of a reliable correctness check is ensured. In particular, voting machines are conceivable

> in which the votes are recorded elsewhere in addition to electronic storage. This is for instance possible with electronic voting machines which print out a visible paper report of the vote cast for the respective voter, in addition to electronic recording of the vote, which can be checked prior to the final ballot and is then collected to facilitate subsequent checking. Monitoring that is independent of the electronic vote record also remains possible when systems are deployed in which the voter marks a voting slip and the election decision is recorded simultaneously, or subsequently by electronic means in order to evaluate these by electronic means at the end of the election day.

The Netherlands: The Netherlands was among a few countries who adopted electronic voting in the early nineties (1990), but it did not go very well in the long run and was abolished in 2008 [Jacobs and Pieters, 2009]. The reason for abolishing the electronic voting was that the voting machines used in elections were susceptible to many attacks, and the results of elections conducted using these machines were not publicly verifiable. Besides, a Dutch public foundation, Wij vertrouwen stemcomputers niet (We do not trust voting computers), demonstrated that e-voting machines used in election leaks enough information to guess the choice of a voter from distance of 20 to 30 meters [1].

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. Moreover, it has already been shown in [Wolchok et al., 2010] that it is possible to manipulate the election results by replacing the parts of machine with malicious look alike components. These components were capable of receiving instruction over wireless communication. As a result, any malicious attacker can control these components from nearby vicinity by sending instructions over wireless channel by using a simple hand-held device and can manipulate the results in their favour [2]. India is mainly criticised for keeping the design of electronic voting machines a closely guard secretes. However, it is not impossible to get access of these machines as shown by [Wolchok et al., 2010]. The worst part, the design of these machines were never audited by any independent third party.

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

[1] https://www.youtube.com/watch?v=B05wPomCjEY
[2] https://indiaevm.org/

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

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

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

Basically New South Wales ran 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/hardware used in the election process were buggy. Furthermore, these software are closely guarded secrets and their source code in not open for general public because of commercial interests of companies [Australian Electoral Commission, 2013] involved in the process. In general, no entity related to government or electoral commission develops the electronic voting software, and predominantly it is outsourced to companies having experience in electronic voting software development. The process of turning a vague idea into a concrete software, also known as software development process, involves requirement gathering, software design, implementation, testing, and maintenance. During this whole process of software development, software testing is the only mechanism for quality assurance. However, testing is not enough for instilling the confidence in software that it is bug free as stated by Edsger W. Dijkstra [Dijkstra, 1972]:

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

Additionally, most of these companies have poor testing practices which exacerbate the software quality problem.

In the next section, given the mission critical importance of electronic voting software I will discuss that software testing in not sufficient for these software, and we should prove the correctness of these software by using formal verification techniques [Beckert et al., 2014]. Furthermore, I will argue that having a formal verification 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 Correctness: Formal Method Approach

Formal verification has been successfully applied in many areas ranging from verified C compilers CompCert [Leroy, 2006], verified ML compiler CakeML [Kumar et al., 2014], Vellvm: Verifying the LLVM [Zhao and Zdancewic, 2012], verified cryptography Fiat-crypto [Erbsen et al., 2019], verified operating system CertiKOS [Gu et al., 2011] and SeL4 [Klein et al., 2009], verified theorem prover Milwa [Myreen and Davis, 2014], verified crash resistant file system FSCQ [Chen et al., 2015], verified distributed system Verdi [Wilcox et al., 2015] , mechanisation of Four Color Theorem [Gonthier, 2008], Fundamental Theorem of Algebra [Geuvers et al., 2002], and Kepler Conjecture [Hales et al.]. One thing I would like to emphasize is that none of these are toy project, and it takes years to develop and verify them. Also, some of these products are used commercially e.g CompCert is used by AIRBUS, and MTU [3] and Fiat-crypto is used in Google's BoringSSL library for elliptic-curve arithmetic [4]. The very basic question to ponder about using formal method to develop software: does it achieve the goal, produces bug free software, given the cost and effort?

I would give two anecdote to answer this question. One of the most basic way to break the software is generating random tests and throwing it to the software under consideration [Miller et al., 1990]. [Yang et al., 2011] developed random C program generator and used these programs to test various compilers. In three years of its usage, they have found 325 unknown bugs in various compiler including GCC [5] and LLVM [6]; 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

[3] https://www.absint.com/compcert/
[4] https://deepspec.org/entry/Project/Cryptography
[5] https://embed.cs.utah.edu/csmith/gcc-bugs.html
[6] https://embed.cs.utah.edu/csmith/llvm-bugs.html

survived 80 million floating point test cases! In the paper, Milestones from the Pure Lisp theorem prover to ACL2 [Moore, 2019], Moore, one of the developer of ACL2, writes:

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

There are numerous instances where formal verification was very useful, and it caught the lurking bugs in design in early stage which could never have been found by testing.

For electronic voting software used in democratic election, where we can not afford to lose a single ballot or miscalculation or any undefined behaviour, should be developed using formal method techniques. In order to ascertain that the formal verification of voting software has been carried out diligently, one therefore needs to

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

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

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

2.1.2 Verifiability: Trust in Electronic Voting

Formal verification is useful in producing the bug free code, but we solely can not establish the trust in the system based on argument of formal verification. The reason is that how would a voter

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

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

Formal verification is certainly a necessary thing in electronic voting because it helps in achieving the correctness of software, but it is not sufficient because it has no mechanism to provide verifiability. Combing both *verification* of the computer software that counts votes, and *verifiability* of individual counts are critical for building

trust in an election process. These two facts, *verification* and *verifiability*, can also be viewed as two sides of a coin from the perspective of two major stake holders of a democracy: i) electoral commission/government, and, ii) voters/participants. Using formally verified software to count the ballots would increase the confidence of electoral commission that he has announced and published the correct result, and the published result always verifies would increase the confidence of voter in the system.

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

2.1.3 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

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

Figure 2.2: Function f computing y on input x

Below is a 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 (trace). 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)
```

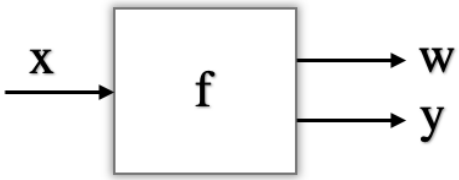


Figure 2.3: Function f computing y and producing witness w on input x

After running the program on the concrete input 34 and 21, it produces the result and certificate (below).

```
gcd 34 21
-------------
gcd 21 13
-------------
     .
-------------
gcd 1 0
-------------
   1
```

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

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

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

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

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

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

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

One may ask the question that can we follow the same approach for vote counting, i.e. unverified counting code, and verified checker? The answer is: it depends. If the verified checker validates the result, i.e. it returns true on the certificate generated by unverified vote counting program then everything is fine from every stakeholders' perspective. However, what about the situation when the verified checker returns false, i.e. invalidates the result. This kind of situation must be dealt carefully by electoral commission, and commission should inspect everything carefully including the vote counting software and various other thing involved in the process. To eliminate this kind of problematic situation, we propose: i) a formally verified vote counting software which produces the result with evidence (certificate), and ii) formally verified certificate checker. The advantage of this approach is that the result produced is always correct (modulo specification), a confidence building measure for electoral commission. Furthermore, the verified checker would always return true on the evidence (certificate) produced by verified vote counting program, confidence booster for voters into the deployed system.

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 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 prover and basic

cryptographic primitives which would enable us later in counting encrypted ballots (without revealing any information about voter's choice).

Chapter 3

Theorem Prover and Cryptography

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

Chapter overview: This chapter is overview of Coq theorem prover and Cryptographic primitives. In the section 3.1, I will give a brief overview of theoretical foundation, calculus of construction (3.1.1) and calculus of inductive construction (3.1.2), of Coq. In the section 3.1.3, I will discuss about `Type` and `Prop` which is very crucial from program extraction point of view. Our goal during this course of formalization was not only proving the correctness of Schulze method, but extracting a executable OCaml/Haskell code to count ballots. In the section 3.1.4, I will focus on dependent types and how it leads to correct by construction paradigm by designing a type safe printf function. Section 3.1.5 focuses on Coq specification language `Gallina` with a example showing that why writing proofs using `Gallina` is difficult and cumbersome, and how it can be eased by using tactics. Finally, in the section 3.1.6, we will take philosophical route to justify that why should we trust in Coq proofs even though they do not appear anywhere near to a proof written by a human.

3.1 Coq: Interactive Proof Assistant

Coq is an interactive proof assistant (theorem prover) based on theory of Calculus of Inductive Construction [Paulin-Mohring, 1993] which itself is an augmentation of Calculus of Construction [Coquand and Huet, 1988] with inductive data-type.

3.1.1 Calculus of Construction

3.1.2 Calculus of Inductive Construction

3.1.3 Type vs. Prop: Code Extraction

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.

explain here the difference between Prop and Types. How it affects the code extraction

3.1.3.1 Reification

3.1.4 Correct by Construction: Type Safe Printf

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

Using the expressiveness of dependent type, we construct a type-safe version of printf. Our goal is to generate compiler error when the given format string and the type of corresponding input values do not match, e.g. printf "%d %s" "hello Coq" 42 should be compiler error because %d is a directive for integer value, but the type of input, "hello Coq", is string. In addition, type-safe printf should print the input when the format string is aligned with type of input, e.g. printf "%s %d" "hello Coq" 42 should print the string "hello Coq 42" because the first directive of format string, %s, and type of input, "hello Coq", is aligned. Similarly, the second directive of format string, %d, is also aligned with the type of input, 42.

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

First, we defined a abstract syntax tree, *format*, to make it explicit the characters we are interested in format string. Additionally, the *format_string* function takes the format string and returns the abstract syntax tree, and the type level, *interp_format*, takes the abstract syntax tree and returns the function type corresponding to format string.

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

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


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

The *interp_format* function returns a function type (`Z -> string -> string -> string`) on the (abstract syntax tree of) format string "%d %s %s"

```
Eval compute in interp_format (format_string "%d %s %s").
(* = Z -> string -> string -> string
     : Type *)
```

Now, we construct a value level function, *interp_value*, whose type is constructed by type level function, and it will take the values to be printed as input. The type of values to print should match the type constructed by type level function for successful type checking otherwise it will be type error.

```
(* value level function whose type is constructed on fly by interp_format
```

```
    function *)
Fixpoint interp_value (f : format) (acc : string) : interp_format f :=
  match f with
  | Fint f' => fun i => interp_value f' (acc ++ of_Z i)
  | Fstring f' => fun i => interp_value f' (acc ++ i)
  | Fother c f' => interp_value f' (acc ++ String c EmptyString)
  | Fend => acc
  end.
```

Finally, we define the printf function, and evaluate it on two inputs: i) printf "%d %s" "hello Coq" 42, and ii) printf "%d %s" 42 "hello Coq".

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

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

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

The first input, printf "%d %s" "hello Coq" 42, is type error because printf "%d %s" returns a value level function whose type is Z -> string -> string, but the type of first argument, "hello Coq", is string which does not unifies with Z, while second one is successfully printed as string.

3.1.5 Gallina: The Specification Language

The example, type safe printf function, 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. Furthermore, It is not suitable for large proof development. In order to ease the proof development, Coq also provides tactics. The user interacting with Coq theorem prover applies these tactics to build the Gallina term which otherwise would be very laborious.

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

```
(* proof written using Gallina *)
```

```
Lemma addition_commutative_gallina : forall (n m : nat), n + m = m + n.
refine
      (fix Fn (n : nat) : forall m : nat, n + m = m + n :=
        match n as n0 return (forall m : nat, n0 + m = m + n0) with
        | 0 => fun m : nat =>
           eq_ind_r (fun n0 : nat => m = n0) eq_refl (Nat.add_0_r m)
        | S n' =>
          fun m : nat =>
            eq_ind_r (fun n0 : nat => S n0 = m + S n')
                     (eq_ind_r (fun n0 : nat => S (m + n') = n0)
                     eq_refl (Nat.add_succ_r m n')) (Fn n' m)
        end).
Qed.

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

3.1.6 Trusting Coq proofs

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

3.2 Cryptography

The word cryptography comes from the two Greek words: *kryptós*, meaning *hidden*, and *gráfein* meaning *to write*. As a matter of fact, in the past, hidden writing (cryp-

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

The modern cryptography originated in 1970 with two ingenious ideas, *Data Encryption Standard (DES)*, and *Diffie-Hellman Algorithm*. Data Encryption Standard, developed at IBM in 1970, is a symmetric key encryption algorithm which uses the same key for encryption and decryption. Since its inception, Data Encryption Standard amassed a bad reputation because of *National Security Agency (NSA)* involvement; however, it had a a practical problem, key management (sort of chicken egg problem). If two parties wanted to communicate securely over insecure channel using Data Encryption Standard, then they needed to agree on a common key. In order to agree on common key, they needed a secure channel where they can securely communicate the key. The solution to this problem came from *Diffie-Hellman* key exchange where two parties can exchange the key securely over insecure channel. Moreover, the advent of *Diffie-Hellman* key exchange started the whole new area of public key cryptography where encryption and decryption key are different. Although *Diffie-Hellman* key exchange suffers from man-in-the-middle (MITM) attack if used for keys exchange in its naivety, e.g. Logjam [Adrian et al., 2015], nonetheless, it is a building block for many other algorithm, e.g. ElGamal Encryption [ElGamal, 1985].

In this thesis, we are mostly concern about public key cryptography and will no longer discuss or explain symmetric key cryptography, hence any mention of cryptography hereafter should be assumed as public key cryptography. The basic working principals of modern day cryptography is based on mathematical principal than vanilla symbol replacement. In addition, it is no longer just used to achieving confidentiality, but various other things, e.g. integrity, authentication, non-repudiation protocol, digital signature, digital cash, etc. These mathematical principal involves various algebraic structures and algorithm to manipulate the object from these structures. For example, the underlying mathematical principal of *Diffie-Hellman* algorithm is hardness of computing discrete logarithm in finite field. .

We will discuss more about these structures in chapter 7

Now we describe the workings of `Diffie-Hellman` [Diffie and Hellman, 2006] algorithm, because all the constructions we have used are based on `Diffie-Hellman` construction. Before we describe the algorithm, we briefly sketch the algebraic struc-

ture Group because it is underlying algebraic structure of `Diffie-Hellman` construction (typically, the underlying structure is multiplicative group of a finite field). Also, note that our definition is influenced theorem-provers/type-theory because we have written the type signature of group operator $*$ and inverse operator *inv*.

3.2.1 Group

A group is a set G, with a binary operator $* : G \to G \to G$, identity element e, and inverse operator $inv : G \to G$ such that the following laws hold:

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

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

3.2.2 Diffie-Hellman Construction

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

1. The two communicating parties, say Alice and Bob, agree with shared public parameters which are finite cyclic group G of order p (p is a large prime) and generator element g.
2. After agreeing with public parameters, Alice and Bob initiates the key exchange protocol (assuming that Alice goes first):
   (a) Alice selects a random element a from the G, computes g^a ($g * g * g ... * g$ a times), and shares g^a to Bob.
   (b) Similarly, Bob selects are random element b from the G, computes g^b, and shares g^b with Alice.
   (c) Finally, Alice computes the key $(g^b)^a$, and Bob computes the key $(g^a)^b$. Because group exponentiation is a commutative operator, both, Alice and Bob, have the common key g^{ab}.

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

DL - Discrete Logarithm problem: An instance of *DL* problem states that given a finite cyclic group G, a generator of g of G, and an element y, finding an element $x \in G$ such that $g^x = y$.

DH - Diffie-Hellman problem: An instance of *DH* problem states that given a finite cyclic group G, a generator of g of G, elements g^a and g^b, finding the element g^{ab}.

DDH - Decision Diffie-Hellman Problem: An instance of *DDH* problem states that given a finite cyclic group G, a generator of g of G, elements g^a, g^b, and g^c, determining if $c = a * b$.

3.2.3 El-Gamal Encryption Scheme

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

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

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

3. **Decryption:** Upon receiving any pair (c_1, c_2), Alice would compute $c_2 * c_1^{-x}$. A basic simplification of $c_2 * c_1^{-x}$ shows that it recovers the plaintext message. The simplification proceeds by replacing the c_2 with $m * h^r$ and c_1 with g^r in $c_2 * c_1^{-x}$. This substitution leads to $m * h^r * g^{-rx}$ which upon further simplification by replacing the h with g^x leads to $m * g^{xr} * g^{-rx}$. Using the same base rule, the term $m * g^{xr} * g^{-rx}$ can be written as $m * g^{xr-rx}$. Since, the underlying group, G, is a abelian group, so we can replace $m * g^{xr-rx}$ with $m * g^0$. The term $g^0 = e$ (the identity of group G) and using the right identity group law, we can replace $m * e$ by m.

3.2.4 Homomorphic Encryption

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

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

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

> $Enc(x) \otimes Enc(y) = Enc(x \oplus y)$ where Enc is encryption function, $\otimes$ is operation on ciphertext, and $\oplus$ is operation on plaintext.

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

$$f(Enc(m_1), Enc(m_2), ..., Enc(m_k) = Enc(f(m_1, m_2, ..., m_k))$$

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

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

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

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

If we mutiply these two cipher together pairwise, we get $(g^{r_1+r_2}, m_1 * m_2 * h^{r_1+r_2})$. After decrypting this combined ciphertext, we will get $m_1 * m_2$. In the

scheme, $\otimes$ is multiplication and $\oplus$ is also multiplication. Furthermore, if our end goal is to achieve multiplication on a bunch of plaintext, then rather than decrypting the corresponding ciphertext individually and multiplying them, we could simply multiply all the ciphertext together and decrypt the final result. The advantage of this scheme is that it does not leak the individual values which, sometimes, is a very crucial property in many application, specifically election voting. In electronic voting protocols, we do not want to reveal the choices of a individual voter, but it is acceptable to reveal the final tally. However, this scheme is not suitable for electronic voting schemes because it is multiplicative. Almost, to the best of my knowledge, all the electronic voting scheme calculate the finally tally by adding the individual choices of all voters, so the requirement is achieve the addition on plaintext. There are many additive homomorphic encryption schemes, e.g. Benaloh cryptosystem, Paillier cryptosystem, etc. In addition, we can modify the ElGamal encryption scheme to make additive. In additive case, it works as:

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

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

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

3.2.5 Zero Knowledge Proof

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

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

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

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

Similarly, we can prove that $((y^{-1} \cdot x^{-1}) \cdot (x \cdot y) = e$. We can also formalize it inside theorem prover and prove it more formally (below is a proof in Coq theorem prover).

```
Lemma inv_distr : forall a b, inv (f a b) = f (inv b) (inv a).
Proof.
  intros a b. symmetry.
  apply inv_uniq_l.
  rewrite <- assoc.
  rewrite  (assoc (inv b) (inv a) a).
  rewrite (inv_l a).
  rewrite (assoc (inv b) e b).
  rewrite (id_l b).
  rewrite (inv_l b). auto.
Qed.
```

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

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

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

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

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

3.2.5.1 Zero Knowledge Proof of Knowledge

Sometimes, the fact that $x \in L$ is completely trivial. For example, for any given finite group G of order p (p is prime), a random element h from the group G, and generator $g$of the group G, a prover claims that there is a x such that $g^x = h$. This is trivial because we know that there always exists such x (discrete logarithm problem); however, the challenge is to show that the prover knows the witness x. Formally, zero-knowledge-proof of knowledge is defines as: let $R = (x, w) \subset LxW$ is a binary relation such that $x \in L$ is common string between prover P and verifier V and $w \in W$, also known as witness, is private to the prover P. Moreover, the goal of prover P is to convince verfier V that $(x, w) \in R$ in zero-knowledge.

3.2.6 Sigma Protocol

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

(a) P sends a message r
(b) V sends a random string e
(c) P replies with z

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

Completeness: when prover and verifier follow the protocol for public input x and witness w then verifier accepts the proof

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

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

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

- Prover P randomly selects an element r from $[0 \cdots \text{q})$, computes $a = g^r$ and sends a to verifier V
- Verifier V randomly selects an element c from $[0 \cdots \text{q})$ and sends it to P
- Prover P sends $z = r + c * w$ to V. V checks $g^z = a * x^c$

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

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

3.2.7 Commitment Schemes

Commitment schemes are cryptographic primitives equivalent to real life sealed lock-box. Once the lock-box is locked and sealed, the content inside it can not be changed without breaking the lock and seal. In general, commitment primitives are backbone of any cryptographic protocol between two parties, communicating over internet, to force them to follow the protocol honestly, even they would have a huge gain from deviating from protocol. For example, in order to save some time before a match, Indian cricket team captain, living in Delhi, and Australian cricket team captain, living in Canberra, decide to toss a coin

in advance over the Internet, using a mobile application called toss-app, for a upcoming series of one-day matches [1]. Assuming the workings of toss-app is naive, i.e. one captain is going to post his call in the chat box, and the other other captain is going to toss the coin at their end and post the outcome in chat box. Furthermore, the decision is taken based on the messages posted by the two captains. In this scenario, we are assuming that the captain, who is tossing the coin, is honest and posting the outcome of the coin honestly in the chat box, which could or could not be the case. The question is can we devise some scheme which would force the both parties to behave honestly? The answer is yes, we can devise such scheme. We would use the sealed lock-box concept, albeit the digital one. Moreover, the first captain would put his call in digitally sealed lock-box and post it in the chat box. Because it is sealed and locked, the other captain would have no idea what is the content inside it. Furthermore, it is impossible to break the lock box, so it is fruitless and waste of time for the other captain to even try. The other captain will toss the coin and post the outcome in a separate digital sealed lock box. Now that we two digital sealed lock box which can only be opened by the respective owners, they would move for the next phase of coin tossing called revealed phase. In the revealed phase, they both would open their sealed locked box to show that what they have locked, and the decision would be taken accordingly [2].

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

(a) Commit phase: sender S commits a value m by generating a random number r and using some algorithm C, which takes the message and random r. Moreover, the committed value produced by the commitment algorithm C, $c = C(m, r)$, is shared with receiver R.

(b) Reveal phase: In the reveal phase, the sender reveals the message m and randomness r which are subsequently used by receiver to verify the result, i.e. the receiver computes $c' = C(m, r)$ and matches it again the given c in the commit phase of protocol.

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

Pedersen commitment: Finally, we give a brief overview of a Pedersen commitment scheme which is based on discrete log. The protocol as follows assuming

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

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

the public parameter available to sender and receiver, i.e. the set up has been conducted to generate the the public parameter, and both parties have these values. These values include a prime p, y a randomly chosen element from Z_p^*, and g a randomly chosen generator from Z_p^*.

- Commit phase: The sender generates a random r from Z_p^*, computes commitment $c = g^r * y^m$ and sent the commitment to receiver
- Verification phase: In verification phase, the sender reveals the original message m and the randomness r. Finally, the receiver computes $g^r * y^m$. If the computed value matches with the commitment received in commitment phase, then she accepts it otherwise reject it.

3.3 Summary

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

Chapter 4

Schulze Method : Evidence Carrying Computation

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

4.1 Introduction

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

Chapter overview: In this chapter, we explain the Schulze method in section 4.2, and its formal specification in the section 4.3. The corner stone of our formalization is a correct by construction dependent inductive data type that represents all correct executions (4.3.1) with the formal proof of that every Schulze election have winners (4.3.2). Every inhabitant of this dependent inductive data type not only produces a final result, but also all the intermediate steps which lead to the notion of evidence or scrutiny sheet (section 4.4). In the section 4.5, we discuss the optimization techniques to overcome the deficiencies in extracted Haskell code from Coq formalization. Based on these optimizations, the extracted Haskell code was able to count millions of ballots in few minutes.

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

Finally, we conclude the chapter in the section 4.6 with the achievements and drawbacks of our work on the scale of *Correctness*, *Privacy*, and *Verifiability*.

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

  $$st(c_0, \ldots, c_{n+1}) = \min\{m(c_i, c_{i+1}) \mid 0 \leq i \leq n\}.$$
- 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 is, always non empty and formally proved in 4.3.2, defined as

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

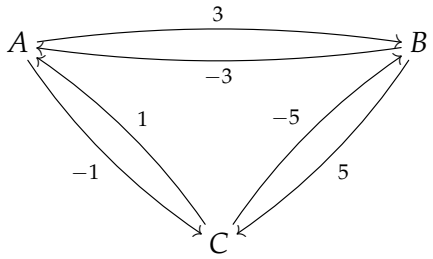
The Schulze method stipulates that a candidate $c \in C$ is a *winner* of the election with margin function m if, for all other candidates $d \in C$, there exists a number $k \in Z$ such that

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

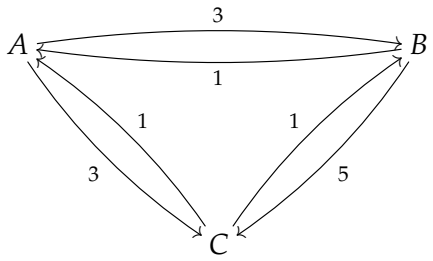
Informally speaking, we can say that candidate c *beats* candidate d if there's a path p from c to d which stronger than any path from d to c. Using this terminology, a candidate c is a winner if c cannot be beaten by any (other) candidate.

4.2.1 An Example

The graph below shows that collective can be cyclic, even if the preferences of individual voters are not cyclic (Condercet paradox).



The main idea of the method is to resolve cycles by considering *transitive preferences* or a generalised notion of margin.



4.3 Formal Specification

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

```
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 candidate, i.e. all the paths from the winner to other candidates are at least as strong as the reverse path. Dually, the notion of loser is that there is a candidate who beats the loser, i.e. the path from the candidate to the loser is stronger than the reverse path.

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

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

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

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

One of the fundamental question about 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*,

and convincing to all, specifically to losers, leaving no ground for speculation. As we stated earlier that in the context of electronic vote counting, this is known as a *scrutiny sheet, or 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.
```

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

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

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

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

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

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

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

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

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

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

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

for $i \geq 0$. It is intuitively clear (and we establish this fact formally) that the iterated margin function stabilises at the n-th iteration (where n is the number of candidates), as paths with repeated nodes don't contribute to maximising the strength of a path. This proof loosely follows the evident pen-and-paper proof given for example in [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 used to establish the type-level winning condition, thus closing the loop to the type level winning condition.

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

Similarly, we connect the propositional losing to type level losing via 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.
```

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

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

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

4.3.1 Vote Counting as Inductive Type

Up to now, we have described the specification of Schulze voting relative to a given margin function. We now describe the specification (and computation) of the margin function given a profile (set) of ballots. Our formalisation describes an individual *count* as a type with the interpretation that all inhabitants of this

type are correct executions of the vote counting algorithm. In the original paper describing the Schulze method [Schulze, 2011], a ballot is a linear preorder over the set of candidates.

In practice, ballots are implemented by asking voters to put numerical preferences against the names of candidates as represented by the 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.

4.3.2 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.

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

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

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

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

4.4 Scrutiny Sheet and Experimental Results

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

result, but instead as a function that delivers a result, *together* with evidence that allows us to verify correctness. In other words, we augment verified correctness of an algorithm with the means to verify each particular *execution*.

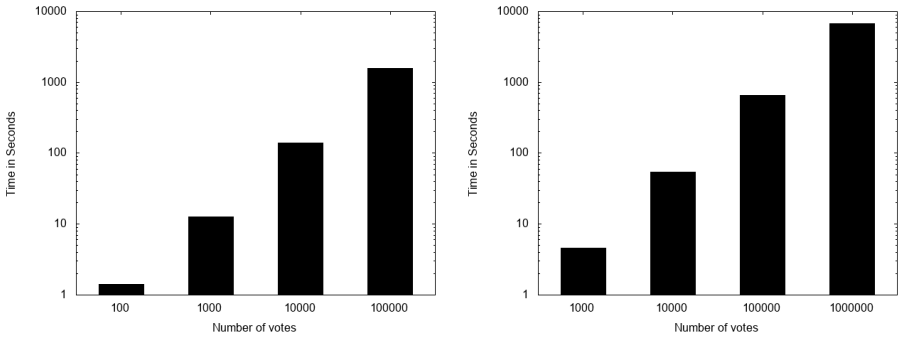
From the perspective of electronic voting, this means that we no longer need to trust the hardware and software (assuming the cast-as-intended and collected-as-cast verifiability) that was employed to obtain the election result, as the generated certificate can be verified independently. In the literature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

Coq's extraction mechanism then allows us to turn our main theorem, `schulze_winners`, 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 4.3.

```
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

(a) Check the correctness of the individual steps of computing the margin function



(a) Direct Extraction (b) Extraction using Haskell Integers

Figure 4.2: Experimental Results (Slow)

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

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

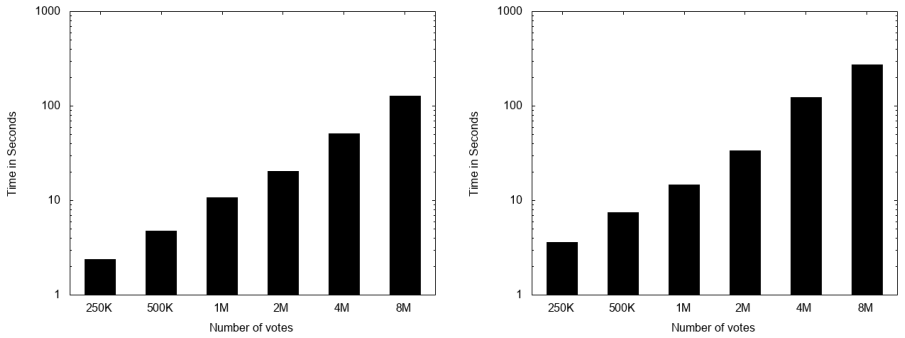
We have evaluated our approach by extracting the entire Coq development into Haskell, with all types defined by Coq extracted as is, i.e. in particular using Coq's unary representation of natural numbers. The results are displayed in Figure 4.2 using a logarithmic scale.

4.5 Counting Millions of Ballots

The previous extracted Haskell code was very slow and was not practical for real life election involving millions of ballots. To scale it to real life election, we 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. However, this efficiency did not come for free, and we had to pay the cost in terms of (almost all) broken proofs. We had to redo all the proofs all over again[1]. Below (Figure 4.3), we include our timing graphs, based on randomly generated ballots while keeping number of candidates constant i.e. 4 (The reason we kept it to

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

4 candidate to show the speed up compared to 4.2. During the experimentation, we ran an election with 21 candidate, and we were able to count 2 million randomly generated ballots before running out of memory.)



(a) Computation of Winners

(b) Computation of Winners and Certificate

Figure 4.3: Experimental Results (Fast)

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.

4.6 Discussion

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

While the Schulze method is not difficult to implement, and indeed there are many freely available implementations on the Internet, comparing the results

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

From the perspective of computational complexity, checking a transcript for correctness is of the same complexity as computing the set of winners, as our certificates are cubic in size, so that certificate checking is not less complex than the actual computation. However, publishing an independently verifiable certificate that attests the individual steps of the computation helps to increase *trust* in the computed election outcome. Typically, the use of technology in elections increases the amount of trust that we need to place both in technological artefacts, and in people. It raises questions that range from fundamental aspects, such as proper testing and/or verification of the software, to very practical questions, e.g. whether the correct version of the software has been run. On the contrast, publishing a certificate of the count dramatically reduces the amount of trust that we need to place into both people and technology: the ability to publish a verifiable justification of the correctness of the count allows a large number of individuals to scrutinise the count. While only moderate programming skills are required to check the validity of a certificate (the transcript of the count), even individuals without any programming background can at least spot-check the transcript: for the construction of the margin function, everything that is needed is to show that the respective margins change according to the counted ballot. For the correctness of determination of winners, it is easy to verify existence of paths of a given strength, and also whether certain sets are 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 4.3) in terms of an already computed margin function that (we hope) can immediately be seen to be correct, and then show that those winning conditions are equivalent to the existence of inhabitants of types that carry verifiable evidence (`wins_type`). We then join the (type level) winning conditions with an inductive type that details the construction of the margin function in an inductive type. Via propositions-as-types, a provably correct vote counting function is then equivalent the proposition that there exists an inhabitant of `Count` for every set of ballots. Coq's extraction mechanism then allows us to extract a Haskell program that produces election winners, together with verifiable certificates.

4.7 Summary

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

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

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

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

Rank all candidates
in order of preference

[4] Lando Calrissian

[3] Boba Fett

[1] Mace Windu

[2] Poe Dameron

[2] Maz Kanata

Chapter 5

Homomorphic Schulze Algorithm : Axiomatic Approach

5.1 Introduction

As we stated in the summary of last chapter that plaintext could lead to privacy problems, e.g. ballot identification (italian attack) Otten [2003]. In this chapter, we will try to achieve: privacy by using encryption, (tallied-as-cast) verifiability by using zero knowledge proof, and correctness of implementation by proving the correctness properties inside Coq theorem prover. One important point to note that we did not formalize any cryptographic primitive inside the Coq, but took an axiomatic approach. Moreover, we assumed the existence of cryptographic primitive and correctness property about their behaviour (axiomatisation of cryptographic primitives). The reason we took the axiomatic approach because our goal was not to formalize cryptographic primitives, but use these primitives to conduct an election which has all three ingredient privacy, verifiability, and correctness. We then obtain, via program extraction, a provably correct implementation of vote counting, that we turn into executable code by providing implementations of the primitives based on a standard cryptographic library (Unicrypt). 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.

Chapter Outline: Todo:Turn this into chapter outline once you finish everything.

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].

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.

5.2 Verifiable Homomorphic Tallying

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

Format of Ballots. Recall that in preferential voting schemes, ballots are rank-ordered lists of candidates. For the Schulze Method, we require that all candidates are ranked, and two candidates may be given the same rank. That is, a ballot is most naturally represented as a function $b : C \to Nat$ 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 $b : C \to Nat$ over a (finite) set C of candidates.

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

We instead represent ballots as 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. The downside of this representation is that it takes $O(n^2)$ space to represent a ballot where n is the number of candidate participating in election.

While the advantage of the first representation is that each ranking function is necessarily a valid ranking and linear space ($O(n)$)) in number of candidates, n, 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

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

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

(a) a shuffle of *encb* together with a ZKP of correctness

(b) decryption of the shuffle, together with a ZKP of correctness

(c) the updated margin matrix, if the decrypted ballot was valid, and

(d) 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

(a) the fully constructed margin, together with its decryption and ZKP of honest decryption after counting all the ballots

(b) 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 5.1 (this implements Item 3c 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 3a 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 3a 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:

(a) generate a (secret) permutation and publish a commitment to this permutation, together with a zero knowledge proof that evidences commitment to a permutation

(b) 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

(c) for each column of the row shuffled ballot, publish a shuffle of the column, also together with a zero knowledge proof of correctness

(d) publish the decryption the ballot shuffled in this way, together with a zero knowledge proof that witnesses honest decryption

(e) 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 sake of completeness, as it is identical to the previous (chapter 4). 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 $\geq k$.

5.3 Formalization in a Theorem Prover

As we stated in the beginning of this chapter that 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.

```
1  Variable decrypt_message:
2    Group -> Prikey -> ciphertext ->  plaintext.
3
4  Variable construct_zero_knowledge_decryption_proof:
5    Group -> Prikey -> ciphertext -> DecZkp.
6
7  Axiom verify_zero_knowledge_decryption_proof:
8    Group -> plaintext -> ciphertext -> DecZkp -> bool.
9
10 Axiom honest_decryption_from_zkp_proof: forall group c d zkp,
11  verify_zero_knowledge_decryption_proof group d c zkp = true
12  -> d = decrypt_message grp privatekey c.
13
14 Axiom verify_honest_decryption_zkp (group: Group):
15   forall (pt : plaintext) (ct : ciphertext) (pk : Prikey),
16   (pt = decrypt_message group pk ct) ->
17   verify_zero_knowledge_decryption_proof group pt ct
18   (construct_zero_knowledge_decryption_proof group pk ct)
19   = true.
```

The difference between the keyword `Variable` and `Axiom` is purely syntactic, and in our case, used as a convenience for extraction (The keyword `Variable` is used if want that variable of function to be lambda abstracted otherwise keyword `Axiom`). In the above, the first two functions, `decrypt_message` and `construct_zero_knowledge _decryption_proof` are *generation* primitives, whereas the function `verify_zero _knowledge_decryption_proof` is a *verification* primitive. We have two coherence axioms. The first says that if the verification of a zero knowledge proof of honest decryption succeeds, then the ciphertext indeed decrypts to the given plaintext. The second stipulates that generated zero knowledge proofs indeed verify.

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

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

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

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

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

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

This is readily translated to the following Coq code:

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

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

- `ecax`: marks the beginning of counting
- `ecvalid`: take a ballot from cast-ballots pile, and the ballot is a valid ballot
- `ecinvalid`: take a ballot from cast-ballot pile, and the ballot is a invalid ballot
- `ecdecrypt`: decryption of fully constructed homomorphic margin from the cast-ballot

- `ecfin`: declaration of winner and loser based on the decrypted margin

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.4 Correctness by Construction and Verification

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

(a) 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).

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

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

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

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

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

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

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

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

The theorems above feature a third type of ballot that is the basis of plaintext counting, and is a simple ranking function of type `cand -> Nat`, and the two hypotheses on the three types of ballots ensure that the encrypted ballots (`ebs`) are in fact in alignment with the rank-ordered ballots (`bs`) that are

used in plaintext counting. The proof, and indeed the formulation, relies on an inductive data type `Count` (section 4.3.1) that can best be thought of as a plaintext version of the inductive type `ECount` given here. Crucially, `Count` is verified against a formal specification of the Schulze Method. Both theorems are proven by induction on the definition of the respective data types, where the key step is to show that the (decrypted) final margins agree. The key ingredient here are the coherence axioms that stipulate that zero knowledge proofs that verify indeed evidence shuffle and/or honest decryption.

5.5 Extraction and Experiments

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

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

As we have explained in Section 5.3, the formal development does not 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.

To fill this hole, we implement the cryptoraphic primitives with help of the UniCrypt library[Locher and Haenni, 2014]. UniCrypt is a freely available library, written in Java, that provides nearly all of the required 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 (4).

Below is 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

```
1  M: AA(13.., 10..) AB(90.., 14..)  AC(11.., 23..) BA(16.., 13..)
2  BB(79.., 46..) BC(12.., 14..) CA(50.., 53..) CB(70.., 68..) CC(23.., 82..),
3  D: [AA: 0 AB: 0 AC: 0 BA: 0 BB: 0 BC: 0 CA: 0 CB: 0 CC: 0],
4  Zero-Knowledge-Proof-of-Honest-Decryption: [..]
5  ------------------------------------------------------------------------
6  V: [AA(42.., 15..) AB(63.., 32..) AC(70, 44..) BA(47.., 34..) BB(16.., 28..)
7  BC(39.., 16..) CA(19.., 13..) CB(57.., 12..) CC(19.., 89..),..], I:  [],
8  M: AA(12.., 11..) AB(13.., 66..) AC(16.., 14.) BA(48.., 31..) BB(15.., 52..)
9  BC(15.., 68..) CA(39.., 69..) CB(12.., 78..) CC(10.., 40..),
10 Row-Permuted-Ballot: AA(53.., 16..) AB(23.., 44..) AC(72.., 47..)
11 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 16GB 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 compu-

tation (which is the same as for the plaintext count) takes negligible time. This is consistent with the mechanism employed by the bindings: each function call from OCaml to Java is inherently memory bounded and creates an instance of the Java runtime, the conversion of OCaml data structures into Java data structures, computation by respective Java function producing result, converting the result back into OCaml data structure, and finally destroying the Java runtime instance when the function returns. While the proof of concept using OCaml/Java bindings falls short of being practically feasible, our timing analysis substantiates that feasibility can be achieved by eliminating the overhead of the bindings.

5.6 Summary

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

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

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

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

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 (4). Given that both parts are practically efficient by themselves, scalability is merely the question of engineering a more efficient coupling.

In a nutshell, the achieved and missed part of this formalization:

- Achieved
  - Correctness: The implementation is formalized in Coq assuming the existence of cryptographic functions and axioms about their correctness behaviour. These primitives were used for constructing evidence, or certificate.
  - Privacy : We don't reveal the content of ballot at any phase of election counting. Therefore, there is no possibility of anyone knowing the choices of a voter other than the voter 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.
- Missed
  - 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.
  - Correctness: We used external unverified libraries for cryptographic code. In general, 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.

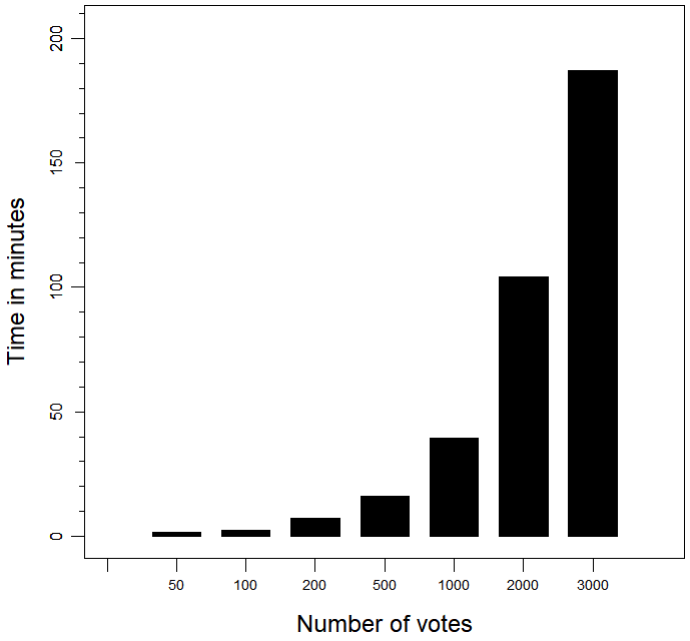
Our formalization leaves some gaps which needed to be filled:

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

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

In the next chapter, we will focus on all the details needed to develop formally verified certificate checker for the certificate we produced in this chapter. However, due to time constraint and complex nature of our certificate, we could not formalize every cryptographic primitive needed to verify our certificate. Rather, we have developed a proof of concept formally verified certificate checker for

IACR 2018 election (International Association for Cryptologic Research), a sim-



Chapter 6

Scrutiny Sheet : Software Independence (or Universal Verifiability)

> Somewhere inside of all of us is the power to change the world.
>
> *Roald Dahl*

The reason for formally verifying the certificate checker is *correctness*. 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.

This thesis builds on and contributes to work in the areas of electronic voting. The original contribution of my thesis is achieving correctness, privacy and verifiability in electronic voting. Although studies in electronic voting has examined verifiability and correctness, there has not been privacy. As such, this study provides additional insight into correctness with emphasis on verifiability by generating a proof certificate (additional piece of data for auditing) . This study is important to my discipline because it bridges the gap between electronic voting and paper ballot election.

But what excites me the most about this study is making the democracy accessible to everyone and making there voice heard.

(a) Need for Scrutiny sheet (Electronic voting)

(b) General Structure of Scrutiny sheet

(c) How to verify it

(d) Extracting a OCaml code

(e) Why should we trust the extracted code

(f) Bits and pieces need to write scrutiny checker

(g) Coq formalization

(h) Explain little bit about Coq

(i) Why do we need scrutiny checker

(j)

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.

I wanted to dedicate this thesis to my first true friend in Australia, Turbo, my house dog who was patient enough to listing my proofs, conjectures and ideas about my PhD, but occasionally bark if it was too much for him.

Some of the great friends who made this journey possible are Caitlin, Milad, Mina, Jim, Ian and Ali, Ahmed. Although, their lunch jokes were too much for me to digest, but I enjoyed it whenever I could. I learned many valuable things from Micheal during the thesis presentation which helped me shaping my personality and how to present it.

I want to thank Thomas Haines for teaching me all the bits pieces of zero knowledge proofs and cryptography. I really enjoyed working with him. One of the best experience of this PhD was travelling to Princeton to participate in summer. Unfortunately, I could not attend Marktoberdorf because it was impossible for me to get the visa of Germany, but I hope things would be more easier in the upcoming future.

Last, but not least, the good wine and beer of Australia could not go unmentioned in this PhD. One of the biggest joy of this PhD was to meet my girlfriend, Mina, who came as a visitor to our logic group.

One of the major disadvantage, as we discusses in the last chapter, of introducing cryptography to make the elections private and verifiable is that now the verification of scrutiny sheet requires specialized knowledge of cryptography to audit the election. In this chapter, we propose that a default checker would increase the number of scrutineers because it can be run by everyone, and the code needs to be inspected by a few. Furthermore, it would help the electoral commission in establishing the confidence in software system.

The key questions to ask: Q. Why do we need scrutiny sheet Because it increases the number of scrutineers

Ballot validity: One of major challenge during the design of homomorphic counting we faced was the suitable data structure for ballot representation. A ballot of form f : cand -> ciphertext would have easier to decide the validity; however, we could not find the homomorphic comparision function to construct the margin matrix without leaking any information. On the other hand, we found that matrix representation of form f : cand -> cand -> ciphertext was very easy for computing the margin matrix, but it was very difficult to decide the validity of matrix because a voter could inflate the ballot by any amount, could create a cycle which does not represent any linear ordering. Our method to decide the validity of ballot, the first thing we needed is the notion of valid ballot. A matrix ballot is valid if it can be represent as linear order (f : cand -> nat), otherwise it is not. The next big challenge was how to convince the voter about this fact with revealing the content of ballot itself.

The very first thing we do is we (as electoral authority) generate a random permutation and create Pedersen's commitment. We keep the random permutation secret and publish the commitment. Now, we take the ballot and permute the each row of it by secrete permutation. We publish the row permuted ballot and zero-knowledge-proof of that published matrix is indeed the row permutation of original matrix. Permutation introduces re-encryption so it is not possible to link the row permuted matrix to original matrix. We do the similar thing with row-permuted ballot and produce the similar evidence for each step.

Informally, the counting starts with making sure that we have taken all the cast ballot under consideration, every entry in initial encrypted margin is encryption of zero, decryption of the encrypted margin which shows that every entry is zero and zero knowledge proof of this fact (constructor ecax). Now, we take a ballot from the pile and check for its validity. If the ballot under the consideration is valid, then we provide the proof of validity and multiply the ballot with running margin matrix (pair wise multiplication), and this becomes our new running margin. If the ballot under consideration is not valid, then we give the proof about its invalidity, move it to list of invalid ballots and our running margin does not change. Finally, we exhaust all the ballots and compute the final margin. We decrypt the final margin with evidence of honest decryption. Based on this decrypted final margin, we computed Schulze winner **??**.

More formally, the constructor of ECount has three piece of information: 1. data related to counting 2. data related to various claims in terms of zero knowledge proof 3. assertion statements

Ecax: counting data: list of uncounted ballots us, encrypted margin encm, decrypted margin decm verification data: honest decryption zero knowledge proof assertions: every entry in decm is zero, and decm is honest decryption of encm

Ecvalid: Counting data: list of uncounted ballot us, the ballot we are counting u, its row permutation v, column permutation of v, w, b that is decryption of w, nm and m verification data: zero knowledge proof, zkppermuv, zkppermvw, zkpdecw, cpi, zkpcpi, assertion data: assertions that v is the row permutation of u by a (secret) permutation whose commitment is cpi, w is column permutation of v by the same (secret) permutation which has been used to permute u to obtain v,

ECinvalid: This is also very similar to ECvalid, except in this case, we don not update the margin

ECDecrypt:

Our final lemma says that if we have a pile of plaintext ballot and the corresponding pile of encrypted of ballots, then the winner produced by one plaintext ballot is same as the winner produced by encrypted. The correspondence between two ballots are shown via a

In final lemma, we establish that if our ciphertext ballot and plaintext ballot are aligned, then will produce the same winner. We prove this fact formally by using the previous chapter counting method, Count. Recall that in previous chapter, our ballot was of form Cand -> nat, while, in this chapter, our encrypted ballot is of form Cand -> Cand ->ciphertext and its decrypted form is Cand -> Cand -> plaintext. We establish that if decrypting the ciphertext ballot and turning it into linear order obtains the same ballot. If we have the two similar isomorphic piles or related pile then we would produce the same winner.

(* This function connects ballot ot pballot *) Definition $map_b allot_p ballot(b : ballot)(p : pballot) := ((existsc, bc = 0)(matrix_b allot_v alidp/ (forallc, bc > 0)(forallcd, (pcd = 1 <-> (bc < bd)(pcd = 0 <-> (bc = bd)(pcd = -1 <-> (bc > bd)$

Our final lemma states that if we take pile of ciphertext ballot, decrypt the whole pile and turn each of them into linear order, then the winner produced by ciphertext would be the same as the winner produced by linear order ballot. It can be thought of other way round as well, but our assumption is that only decryption is deterministic so we did not say that take linear form ballot, turn it margin matrix and encrypt the margin matrix to obtain the ciphertext. This would require the assumption that encryption is deterministic which is not true for ElGamal Encryption (We geared towards ElGamal). Also, in general, encryption is not deterministic.

Recall that in chapter 3, our ballot was of Cand -> nat, while the pballot is form Cand -> Cand -> Z. The function $map_b allot_p ballotconnectsthesenotions.Itsaysthenwheneverbisinvalidthenpinvalid, a$

(* This reason I am going with this proof is that my proofs depends on this. But inductive type is more elegant *) Fixpoint $mapping_b allot_p ballot(bs : listballot)(pbs : listpballot) : Prop.Proof.refine(matchbs, pbswith|[], [] => True|[], h ::_=> False|h ::_, [] => False|h1 :: t1, h2 :: t2 => map_b allot_p balloth1h2/ mapping_b allot_p ballott1t2end); inversionH.Defined.$

We show that the winner produced is same if the plaintext ballot and ciphertext ballot are related. The meaning of relatedness here is that every ballot in plaintext is decryption of ciphertext. More formally, a ciphertext ballot eb is decrypted as plaintext ballot pb which, in turn, is reflected as a function f : cand -> nat.

The proof of the lemma hinges on the fact that margin created by both piles would be same. Once we establish the connection that margin is same then the proof onwards is same. Because the margins are same so they would produce the same winner.

We have also proved this formally that by using the lemma Lemma same$_m$*argin*$_e$*nc* : $forallbsinbsinbs0encmencm0ss0(c0 : ECountgrpbss)(c1 : ECountgrpbss0), s = epartial([], inbs)encm-$ $s0 = epartial([], inbs0)encm0- > forallcd, decrypt_message grpprivatekey(encmcd) = decrypt_message grpprivatekey(encm0cd)$.

6.1 Ecount explanation

The first constructor, ecax, bootstrap the counting process. It has two part, the data part and assertion on data to ensure the correctness or integrity of process. Ecax consist of intial uncounted ballot, us, initial encrypted zero margin, encm, decrypted zero margin, decm, and honest decryption zero knowledge proof deccZkp. The first constraint us = bs establishes the fact that we started with all the ballots which were cast, the second line (forall c d, decm c d = 0) states that every entry in decm is 0, and (forall c d, verify$_d$*ecryptionzeroknowledgeproofgrp*$(decmcd)(encmcd)(zkpDeccd) = true)emphasizeth$

The second constructor ecvalid continues with counting if the ballot under consider was valid.

Why do we need scrutiny sheet? We need verifiability in electronic setting everyone treats it as black box but wee need more W e need to make sure that our election is verifiable . For that reseaon we need scrutiny seet . Scrutniy sheet con tains enought information to audit the election . We ut information in the seet which needs to be checkerd by scrutineers . Then we xtract code from it and verify the election based on sheet., We need Group theory and constructive math (Coq) to make sure that we reach the goal . We need to scrutiny the election . to make sure that it has establisthed the trust . We will formalized it Coq theorem rover a to make sure that we have roverd all the properties of electrionic voitng method

Scrutiny sheets are needed for electronic voting because it attests the need for verification . It puts the confidence in the system and published results. It may needed by the voters of democracy and third party auditors to attests the outcome of election . What do we need in scrutiny sheet ? Enough information with the confidence that it has been produced correctly. Wee.ll. the whole point is audit the election so why do we need confidence ? because we don't know if machine is behaving correctly, or the bulletin board is galloping the ballots ? Once we have made it sure that there won't be any fishy things then we can make sense if elections. We need to inform the voter about the process and mathematics behind the cryptography which off course is difficult but every democracy has some voter who can do this. Publishing the result to

make sure that it checks out is really great benefit for democracy. Once the user or voter knows that end to end verifiability hold then he can ascertain the results are correct . In the scrutiny sheet we have encrypted ballot , encrypted zero margin , and zero knowledge proof for the fact that our maring is zero . Then we take a ballot and multiply it encrypted zero margin

A Scrutiny sheet (Theme) is an essential ingredient for verifying an election (Rheme) (Level 5). (Rheme) Verification is the process of establishing the truth of fact, and (Theme) scrutiny sheet facilitates this process in a democratic election (level 5). The published (theme) scrutiny sheet make sure that the outcome is indeed the honest one, and everyone can establish or ascertain the outcome of election based on data published in scrutiny sheet (level 4). Some politicians question the value of (Theme) scrutiny sheet, but more importantly, why should we put our trust in scrutiny sheet? (level 4) Scrutiny sheets have been put forward by the electronic voting research community to achieve the idea of end-to-end verifiability (level 4), i.e. we do not need to trust the software/hardware in election process to trust the result (level 4). Many communities over the Internet publish the scrutiny sheet of their election **??**.

6.2 Components of Scrutiny Sheet

Our scruntiny sheet contains data about various facts. These data connects the dots and leave no gaps to make the process of verification robust.

- In the beginning of scrutiny sheet, we have all the uncounted ballots, encrypted zero margin, zero-knowledge-proof that the encrypted margin is indeed encryption of zero
- In next line, we take a ballot and depending on its validity, we either update the margin matrix which amounts to multiplying the ballot under consideration with entries in margin matrix, or we discard the ballot and move it list of invalid ballots. In this case, margin matrix remains unchanged
- This process is very complex and we need to make sure this step is verifiable. In order to do so, we augment the scrutiny sheet with extra data (zero-knowledge-proofs) so that it can be later verified.
- How do we decide if a ballot is valid or invalid? A ballot is represented as linear then it is valid, i.e. we can find a function which ranks the candidate linearly, otherwise it is not valid. In order to decide the validity of a ballot, we generate a random permutation, and commit it using Pedersen's commitment, and publish the commitment.
- The advantage of this is that no one can break the Pedersen's commitment given the polynomial time attacker and Diffie-Hellman assumption holds
- We also can not open the it any other way than we committed to which forces to have binding property

- Now that we hiding and binding property, we take the ballot and permute each row of it by the permutation we committed. We publish the zero-knowledge-proof for this fact that the row permuted matrix is indeed the permutation of ballot by some permutation whose commitment is published
- We take the previoulsy row permuted ballot and permuted each column of it again by the same permutation with similar zero-knowledge-proof
- In the end, we decrypted the column permuted ballot and publish the it. Any one can see if this ballot is valid or not by making sure that if it is linear function or not and connecting the dots backward. One key point of this whole process is that we never reveal the permutation otherwise anyone could construct the origin ballot from decrypted permuted one. In order to be verifiable, we publish the zero-knowledge-proof that the ballot is indeed permuted by the same permutation whose commitment we have published
- One important point of this whole process is that we never reveal our permutation, otherwise

software independence(Rheme), i.e. we do not need to trust the software involved in election process to trust the result. (Rheme) Software independence is a weaker notion of end-to-end verifiability. End-to-end verifiability basically makes the whole process independent of hardware and software involved in the process of election voting. We no longer need to trust the

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

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

- 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.

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].

(a) first paste the certificate
(b) take each piece of information, and show that how can it be verified
(c) Flesh the details of honest decryption zero knowledge proof
(d) Flesh the details of permutation shuffle

(a) Sketch the Monoid -> Group -> Abelian Group -> Field -> Vector Space
(b) Sketch the Schnorr Group (p = q * r + 1)
(c) Sketch the Sigma Protocol,
(d) Write Elgamal encryption, decryption, and cipher text multiplication
(e) Show the honest decryption (I know the private key using Discrete logarithm problem. Proof of Knowledge in Zero Knowledge)
(f) Maybe some commitment scheme

Now that we have explained the every bit of information about our certificate, we flesh out the details of algebraic structure we need to develop the formally verified certificate checker. We develop the notation of Monoid

Class Monoid (S : Type) (f : S -> S -> S) (e : S) := (Associativity: forall a b c, f a (f b c) = f (f a b) c) (IdentityLaw : forall a : S, f a e = f e a = a)

A group is monoid with each element have a inverse

Class Group (S : Type) (f : S -> S -> S) (e : S) := (Monoid S f e) (InverseLaw : forall a, exist b, f a b = e)

An Abelian Group, in honor of Henrik Abel, is a Group with commutativity law

Class Abelelian-Group (S : Type) (f : S -> S -> S) (e : S) := (Group S f e) (CommutativeLaw : forall a b, f a b = f b a)

A Field is a set with two binary operation f and g

Class Field (S : Set) (F+ : S -> S -> S) (F* : S -> S -> S) := Here goes field

Class Vector Space := Here goes vector space

Now we describe the sigma protocol briefly. A sigma protocol is efficient zero-knowledge-proof protocol in which we a prover "peggy" conveys the truth of statement to the verifier "verity" without revealing anything other than the truth itself. The requirement of any protocol to be zero-knowledge is: i) Soundness ii) Completeness, and iii) Zero knowledge. (verifier learns nothing other than the truth of statement)

One of the interesting fact about this notion we define the absence of knowledge without defining what the knowledge is. Now dive deep into representing sigma protocol (a convenient way to do zero-knowledge-proof) inside a theorem prover

Class SigmaProtocol (G : Group) (H : Group) (f : G -> H) := Here goes the details.

In sigma protocol, the prover generates a random value and commits it using Pedersen's commitment. He send the value to the verifier. Verifier generates a random number in a given Group Gp and send it back to verifier. Verified sends the r * c + q to prover and prover accepts if the transcript is valid otherwise he rejects it.

Now we prove that Diffie-Hellman discrete logarithm problem is indeed the instance of sigma protocol, i.e. it follow the soundness (special), completeness and zero knowledge property.

We have to show three things:

- The diffie-hellman implements correct encryption
- Honest decryption zero knowledge proof
- Pedersen's commitment
- Wikstrome commiment consistent shuffle

In this chapter, we formally prove the diffie-hellman, honest decryption zero knowledge proof, and Pedersen's commitment. However, we have not verified the Wikstrome's commiment consistent shuffle because it involv

Electronic voting is nightmare and the reason is that a simple possible of going anything wrong in the components used to conduction electronic voting would contribute significantly to result, possibly inverting it. We can divide the world on the scale of electronic voting: 1. Countries who abolished it 2. Countries who are continuing it 3. Countries who are running a pilot project

The two early adopters of electronic voting are Germany and The Netherlands. It was abolished in Germany by Supreme court because the results produced were not verifiable. German Supreme court said in her statement that we are not against the electronic voting, but electronic voting violates the fundamental rule of German constitution that every democratic election to public office should be verifiable. German court emphasized on public examinability.

In German elections, the vote counting software treated as a black-box. It will take a pile of ballots and produce the result with any evidence if the result produced is indeed the correct one. This situation is in contrast with paper ballot voting where every single step in the process is verifiable.

The nature of data in electronic voting makes the problem very difficult. Everything piece of information is electronic which could be manipulated very easily without anyone noticing. The other problem is that the amount of effort it takes to change one ballot is almost same as the ballot from whole country. This is indeed a big problem, because there are many corporates who would like to elect some favourite party and given that the democratically elected government controls all the resources, this would lead to a chaos.

The nature of data in paper ballot is not electronic which makes it not susceptible to many electronic attack, e.g. it would always take more effort to vote manipulating at mass scale; however, it is very unlikely to go undetected.

Electronic voting has many benefits and we can solve the problem then we can reap the benefits of it. Electronic voting is always a active area of research and research community came with the rule which every voting protocol should have: 1. Correctness: The results produces should be correct, and convincing to everyone, specifically to losers, without any doubt 2. Verifiability: The produced result should be verifiable by any one. Verifiability is easy to achieve in plaintext ballot counting, but it is very difficult problem in case of encrypted ballot. In order to achieve verification in electronic voting, we have used zero-knowledge-proof.

3. End-to-end verifiability 3.1: Cast as intended 3.2: Collected as cast 3.3: Tallied as cast

End to end verifiability is a very strong notion and it makes the produced result by independent of software used to product it. Any bug in the software or hardware would not affect the result and it can be caught easily. Cast as intended: Cast as intended can be used to audit the voting client software used at polling booths to collect the ballot. It was first proposed by Josh Benaloh to catch if machine is behaving maliciously. The process is a voter would cast his options and voting client would encrypt the ballot. At this point, a voter has two option. He can either commit

his ballot which would be added to bulletin board, or he can challenge the machine to decrypt the ballot which would make the ballot invalid. If machine is cheating then it will caught with very high probability, because machine would have no idea when the voter would commit to his options.

Collected as cast: Once every ballot have been cast and posted at bulletin board then each voter would identify his/her ballot (Question to myself: How would this solve the problem of ballot stuffing. Everyone has to collaborate

1. Each voter checks if their ballot is appearing at bulletin board 2. Only eligible voters have cast their vote (under no circumstance, there should be more votes than eligible voters))

3. Tallied as cast: It emphasize on that produced result should be based on the ballots at bulletin board.

This thesis is concern about privacy, verifiability, and tallied as cast. Our approach is that a vote counting software should not only produce the result, but it should produce a independently checkable certificate. The purpose of this certificate to make sure that any independent third party can ascertain that the produced results are indeed the correct one.

As long as the cast-as-intended and collected-as-cast has been followed, then our counting software produced the result as collect-as-cast with a witness to attest it result.

Question: 1. Why do we need scrutiny sheet. 2. Why do we need a formally verified scrutiny sheet checker

One the short comings our method, as we discussed in the previous chapter, was that it requires intricate knowledge of cryptography to audit the election which substantially reduced the number of scrutineers. While it is not very difficult to find cryptographer in any economy to audit it, they are not in abundance and also they are not representative of democracy. In order to increase the number scrutineers, we followed the route of proving a default checker which anyone can run on the election data and see if the result checks out. The reason for formally verifying it to make sure that it is bug free and does not deviate. Consider the scenario where published certificate checker does not return true on certificate because of its own bug, but not because the certificate was produced incorrect. This would off course be a devastating situation, hence the reason for formally verifying it.

We would follow the same argument. We would write the checker and formally verify it followed by publishing the source code so that any one can have a look and see if the verification has been carried out diligently.

(Question: It going in circular fashion that how can we expect to find more logicians that cryptographers)

The asture reader can point that now we have shifted the burden from cryptographers shoulders to logician shoulders. We agree that this is indeed the case, but what make it worthwhile that we have given the default checker which is verified,

i.e. bug free. What would happen if we do not provide the checker, then how likely it would be for community/voters to develop the verified checker? It would off course be healthy for democracy if some one write his own checker and can match the result produced by his checker against the default one.

Many researcher has formally verified the electronic voting scheme in symbolic model (What is symbolic model), however, to the best of our knowledge, this is first study of electronic voting algorithm implemented inside a theorem prover and its properties were proved correct.

The closest of our work is Dirk Pattinson, Milad -Ketabi, but none of these researcher have taken the privacy into consideration. Their counting algorithm is simple (debatable?) and counts planetext ballot, while, our implementation can count the encrypted ballot. Our project progresses from counting a few hundred thousand planetext ballots to millions of planetext ballot to few thousand encrypted ballot (we will talk more about inefficiency in chapter 3). While Milad's work has produced the certified checker in CakeML, but it, again, no longer considers the scrutiny sheet produced by counting encrypted ballots augmented with zero-knowledge-proofs. In order to write a verified checker for these kind of scrutiny sheet, it takes non-trivial amount of knowledge and effort.

Cryptography is all about hiding secrets by using knowledge of number theory. The first public key cryptography was devised by Martin Hellman based on hardness of discrete logarithm in Group. Public key cryptography involved public key, private key, encryption function and decryption function. Except private key, every thing is public knowledge. If Alice want to communicates with Bob, then by some means, using email, or going to his website, she would fetch the public key of Bob. Using the public key of Bob, she would encrypt the message that she wants to convey by using encryption function and send it to Bob. This encrypted message would be visible to everyone including Bob; however, it only possible for Bob to recover the message by his private key (related to his public key). (Before this introduce the notion of attacker). Given that encrypted message is available to everyone, is it possible that some one other Bob can recover the original message? Depending on encryption, if it is RSA or Diffie-hellman, it always hard for polynomial time attacker to recover the message. Re-iterate: It is not impossible to recover given infinite amount of time, but hard. The formal notion of hardness in cryptography is borrowed in computability theory.

Homormorphic encryption: This problem was first proposed by Ron Rivest, and the purpose of homomorphic encryption is to run computation on encrypted data. If we encrypt the a plaintext message using Diffie-Hellman algorithm: E(m, prkey) = $(g^r, g^m * g^r) What would happen if we multiply two ciphertext? E(m1, prkey) = (g^r 1, g^m 1 * g^r 1) E(m2, prkey) = (g^r 2, g^m 2 * g^r 2)$

Multiplying the correspoing elements of tuple, $(g^(r1 + r2), g^(m1 + m2) * g^(r1 + r2)), and let's decrypt it g^(m1 + m2)$.

(a) Talk about Additive Homomorphic

(b) Talk about Multiplicative Homomorphic

(c) Talk about Full Homomorphic

As we can see that multiplying a ciphertext amounts to multiplying the corresponding plaintext and encrypting them. It was an open problem for almost 40 years that if there is It is also an active area of research by Craig Gentry work on first fully homomorphic encryption. In our implementation, we needed additive homomorphic encryption and our implementation is tailored towards ElGamal Encryption algorithm, but it would work with encryption scheme which is homomorphically additive.

Commitment Schemes: (* Question: What is commitment scheme and why do we need it our formalization *) Commitment schemes are basically digital implementation of "locked box". Two parties who does not trust each other are participating in some sort of zero-sum-game, e.g. gambling. Both parties toss a coin simultaneously, and now they want to reveal their outcome. Who ever first reveal his/her option would lose the strategic advantage because the other party would change his/her coin to win the game. In order to solve this problem, We use the digital lockbox called commitment. Commitment schemes are cryptographic way to commit to some value. Commitment scheme has two properties: 1. Hiding, no one can guess the original value based on the committed value 2. Binding, committed value can not be open in two different ways

Now, we take the same situation, but augmented with commitments. First party tosses the coin and publishes the committed value. Second party tosses the coin and publishes the committed values. Because of hiding property, either parties can not guess the original value of each others based on the committed values, and because of binding property, either party can not open their value in any other way other than the original one. There are various commitment schemes, but two most popular are 1. Hash based commitment scheme 2. Discrete Logarithm Based

Hash based commitment: First party tosses the coin and generate a random value r. It computes hash of outcome of the coin toss appended with randomness r and publishes it. Similarly, the second party also follows the similar steps and publishes its commitment. Later, both parties open their coin toss with randomness to verify each others commitment.

Discrete Logarithm: Public setup: (p, q, g, y) First party tosses the coin and generate a random value r. It computes $g^r * y^m where is outcome of coin toss and r is the randomness and publishes it. S$

One advantage of discrete logarithm commitment schemes over Hash based is that homomorphic property. Hash based scheme can not be exploited for homomorphic property, while discrete logarithm can be.

6.3 Certificate : Ingredient for Verification

There are two notion of verification. Software verification and election verification. A

6.3.1 Plaintext Ballot Certificate

Flesh out the details needed here for writing proof checker

6.3.2 Encrypted Ballot Certificate

Flesh out the details needed here for writing proof checker

6.4 Proof Checker

Proof checker or auditor for verifying the encrypted ballot scrutiny sheet requires: 1. Checking if the encrypted margin is indeed the encryption of zero. In order to verify this claim, he needs to check the zero-knowledge-proof of honest decryption **??** 2. For each ballot, he needs to check our claim of a ballot is valid or not. For that he, needs to check the each row of ballot is indeed permuted by the permutation whose commitment (Pedersen's) has been published. **??**. We have already published the commitment of (secret) permutation with the zero-knowledge-proof of shuffle. Then, he needs to check the honest decryption of row permuted shuffled ballots. Similarly, he needs to checks the similar facts for column permuted ballot. Next, depending on the claim if it is valid then he needs to check if the new margin matrix is equal to previous margin matrix multiplied by ballot under consideration. If it is not valid then he needs to check if the new margin is same as previous margin and the invalid ballot has been moved to pile of invalid ballots. He needs to do this until ballots are exhausted. After ballots are exhausted then we need to check the decryption of final margin. We have already explained the rest of the process in section **??**.

Flesh out the details: 1. Why do we need to to prove the properties. Because we wanted to show that our implementation is indeed the implementation of Schulze method (weak argument).

6.5 Machine Checked Proofs

This chapter is still ongoing investigation. In this chapter, we are going discuss some of the properties of Schulze method, and proving that our implementation of Schulze method follows this property.

1. Condercet winner: Schulze method follows the Condercet criterion. Condercet criterian is that if a candidate beats everyone in head to head count, then he should be the winner. We draw the upon on the definition of constructed margin and define the Condercet winner as:

Definition condercet (c : Cand) (marg : Cand -> Cand -> Z) := forall d, marg c d >= 0

The lemma we prove that condercet winner wins the election is:

Lemma condercet$_w$inner$_i$mplies$_w$inner$(c : cand)(marg : cand- > cand- > Z)$: condercet$_w$innercmarg$-> c_w$insmargc $= true$.

The proof hinges on the fact that if a candidate 'c' beats every other candidate in head to head count, then it beats everyone in generalized margin. Lemma condercet$_w$inner$_m$arg$(c : cand)(marg : cand- > cand- > Z) : foralln,(foralld, margdc <= margcd)- > foralld, Mmargndc <= Mmargncd$.

Recall that our winner definition is exactly the same, i.e. winner beats the everyone based on generalized margin. c$_w$insc $:= foralld, Mcd >= Mdc$.

2. Reversal symmetry: Reversal symmetry states that if we reverse the options on ballot, then winner should not winner any more. Schulze method follows the reversal symmetry. To prove reversal symmetry, we assume that when we reverse the ballot then constructed margin would minus one multiplied by margin constructed from original ballot (Not very clear). First we defined the notion of unique winner Definition unique$_w$inner$(marg : cand- > cand- > Z)(c : cand) := c_w$insmargc $=$ $true/\ (foralld, d <> c- > c_w$insmargd $= false)$.

and reverse margin

Definition rev$_m$arg$(marg : cand- > cand- > Z)(cd : cand) := -margcd.(*Wemultiply-$ 1$tomarginmatrix*)$

Our main lemma about stating the reversal symmetry is

Lemma winner$_r$eversed : $forallmargc, unique_w$innermargc$- > (existsd, c_w$insmargd $=$ $false/\ c <> d)(*atleastweneedtohavetwocandidates*)- > c_wins(rev_m$argmarg$)c =$ $false$.

Definition reverse$_{ab}$allot$(p : ballot) := letlt := mappcand_allinletw = zipcand_all(revlt)infunc =>$ $search_{ci}n_l istw$

It says that if we have a unique winner with respect to margin then the unique winner is loser with respect reverse margin.

3. Monotonicity Monotonicity criterion is that if we increase the ranking preference then it would not change the winner. The proof for this is very simple. We need to notice the increase the margin would not change the constructed margin.

Definition inflate$_t$he$_b$allot$_b y_k(k : nat)(f : cand- > nat)(c : cand) := k + fc$

Prove that

forall p m k, update$_m$arginp$(m : cand- > cand- > Z) = update_m$argin$(inflate_t he_b allot_b y_k kp)m$

It shows that the constructed margin would be the same as ballots inflated by some natural number k.

4. Majority rank$_c$andiate$(cd : cand)(p : ballot) := beq_nat(pc)(pd)c_rank_h igher(c :$ $cand)(p : ballot) := forallb(fund => pc > pd)(removeccand_all)$.

Fixpoint Majority$_c$riterion$(c : cand)(l : listballot)matchlwith|[] => 0|h :: tl =>$ $ifc_rank_h ighercpthenS(majority_criteronctl)else(majority_criteronctl)end$.

5.

6.5.1 A Verified Proof Checker : IACR 2018

Write some details about proof checker.

6.6 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.

6.7 Software Bug

Despite all these advantages, electronic voting is nightmare because the minuscule possibility of a bug in the software used for voting could lead to a disaster, possible inverting the result. The nature of (electronic) data and ease of its manipulability/misinterpretation bring electronic voting many problems, which are not present in paper ballot election, that makes it perfectly susceptible to delivering wrong and unverifiable result. If a software program used in electronic voting for reading the ballots has byte order bug, or even if it depends on some other software which has byte order bug, then the interpretation of ballot would completely be different from the what the voter had in mind. More often than not, these software programs are configured incorrectly and run at the top of (untrusted) operating system and hardware. Operating systems have millions of lines code (Linux has 15 millions lines) which exposes a large attack surface and could be exploit, possibly by current government or foreign country, for illegal gain. The worst, these software and hardware used in electronic voting process are commercial in confidence and treated as a black-box, and, most often, their source code or design is not open for public scrutiny. These software program take a pile of ballot and produce the result without producing any evidence about the correctness of result. As a consequence, from casting the ballot electronically and declaring winner based on cast (electronic) ballot lacks, basic assumption of democracy, correctness, privacy and verifiability.

These software program, during their operations, in electronic vo

The software and hardware used in the electronic voting process is 1. One major reason for this is the nature of data in electronic voting. It can be easily manipulated without anyone noticing it. 2. The software and hardware used in electronic voting

Electronic voting is getting popular in many countries, and the reason for it's popularity is cost-effectiveness, faster results, and high voter turnouts. Electronic voting machines (EVM) have clear benefits for large, populous countries like India, with 900 million eligible voters. In the 2019 general election, India had a 67 percent (roughly

600 million) voter turnout and with the help of EVM, was able to conduct the election in 2 days. Australia has a smaller population, but its massive size and sparsely populated land makes an election a big logistical challenge. Australia is actively pursuing Internet voting to ease the logistic challenges and engage more citizens from the remote places. For over a decade, EVM has been integrated into a democratic Estonia since 2005, whuch demonstrates that the technology is relative mature. The first country to adopt internet voting and showing confidence in technology. In 2019 elections, as claimed on e-estonia [Est], the time saved was 11,000 working days.

A democracy can be described as a system where every eligible participant has equal right to express their opinion(s) on different matters. One of the most important example of expressing the opinion is holding elections to elect the leader of country. During the elections, every eligible participant expresses their opinion on a paper, also known as ballot, by ranking the participating candidate in according to their preference. Later, once the cast finishes, a candidate is elected from participating candidates by combing the all choices of participant. The paper ballot method works great, except it is very time consuming, expensive and error prone. Moreover, paper ballot elections are not environment friendly, and it produces a lot of waste in form of discarded papers. In order to solve the problems posed by paper ballot, many countries are adopting electronic voting. Electronic voting is getting popular in many countries, and the reason for its popularity is cost-effective, faster result, and high voter turn out. Undeniably, electronic voting has helped Australia to ease the logistic challenges of elections because of its massive land size and sparsely population and save millions of dollars. it has helped India, the second most populous country with 900 million eligible voter, to declare 2019 election with 67 percent voter turn out (roughly 600 million) in 2 days. Estonia, a labour shortage country, has saved thousands of man hours, 11,000 working days, by using electronic voting. Despite these successes, EVM is still not adopted as widely as it could be.

The slow adoption of EVM is largely because of fears about the accuracy of systems. Even though there is a minuscule possibility of a bug in the software used for voting, the consequences of problems like this can be disastrous. If a bug damaged, or even inverting the results. The fear is that problems like this could lead to civil unrest and economic consequences. The nature of (electronic) data and ease of its manipulability/misinterpretation bring electronic voting many problems, which are not present in paper ballot election, that makes it perfectly susceptible to delivering wrong and unverifiable result. Assuming that a software program used in electronic voting for reading the ballots has byte order bug, or even if it depends on some other software which has byte order bug, then the interpretation of ballot would completely be different from the what the voter had in mind. More often than not, these software programs are configured incorrectly and run at the top of (untrusted) operating system and hardware. Operating systems have millions of lines code (Linux has 15 millions lines) which exposes a large attack surface and could be exploit, possibly by current government or foreign country, for illegal gain. The worst, these software and hardware used in electronic voting process are commercial in confidence and

treated as a black-box, and, most often, their source code or design is not open for public scrutiny. These software program take a pile of ballot and produce the result without producing any evidence about the correctness of result. As a consequence, from casting the ballot electronically and declaring winner based on cast (electronic) ballot lacks, basic assumption of democracy, correctness, privacy and verifiability.

Despite all these benefits, electronic voting is nightmare because the minuscule possibility of a bug in software used in voting could lead to a disaster, possibly inverting the results [Lewis et al.], [Halderman and Teague, 2015], [Aranha et al., 2019], [Feldman et al., 2007]. By its inherent nature electronic voting has many problems, which are not present in paper ballot elections that makes it perfectly susceptible to delivering wrong and unverifiable result [Wolchok et al., 2010]. The software and hardware used in the electronic voting process are treated as a black-box and commercial in confidence [Australian Electoral Commission, 2013] which violates the fundamental property of public examinability of any democratic election. More often than not, these software programs are configured incorrectly [Kohno et al., 2004] and run at top of (untrusted) operating system and hardware. Operating systems have millions of lines of code (Linux has 15 million code) which exposes a large attack surface and could be exploited for illegal gain in election, possibly by current government or foreign country. As a consequence, from casting ballot electronically to declaring results based on electronic data (ballot) may raise several questions:

The key ingredients for achieving the secure election is privacy and verifiability. Achieving either is trival; however, their combination is notoriously hard problem. We can achieve the combination of both by sophisticated cryptography. We can encrypt the ballot to achieve the privacy and use zero-knowledge-proof to assets the different claims made during the process.

Chapter 7

Machine Checked Schulze Properties

Not planned but hopefully it will done

7.1 Properties

List some properties which it follows with pictures ?

7.1.1 Condercet Winner

7.1.2 Reversal Symmetry

7.1.3 Monotonicity

7.1.4 Schwartz set

Chapter 8

Conclusion

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

8.1 Related Work

Here goes the details for related work

8.2 Future Work

Possibility of future work

8.2.1 Formalization of Cryptography

8.2.2 Integration with Web System : Helios

Bibliography

e-governance. https://e-estonia.com/solutions/e-governance/i-voting/. Accessed on October 17, 2019. (cited on pages 1 and 93)

Electronic Voting. https://www.e-voting.cc/en/it-elections/world-map. Accessed on October 17, 2019. (cited on page 9)

German Constitution. https://www.bundesverfassungsgericht.de/SharedDocs/Entscheidungen/EN/2009/03/cs20090303_2bvc000307en.html. Accessed on October 18, 2019. (cited on page 10)

New South Wales Election. https://www.abc.net.au/news/2015-02-04/computer-voting-may-feature-in-march-nsw-election/6068290. Accessed on October 18, 2019. (cited on page 12)

Western australia senate election. https://www.theguardian.com/world/2014/feb/28/western-australia-senate-election-re-run-to-be-held-on-5-april. Accessed on October 18, 2019. (cited on page 9)

Adrian, D.; Bhargavan, K.; Durumeric, Z.; Gaudry, P.; Green, M.; Halderman, J. A.; Heninger, N.; Springall, D.; Thomé, E.; Valenta, L.; VanderSloot, B.; Wustrow, E.; Zanella-Béguelin, S.; and Zimmermann, P., 2015. Imperfect forward secrecy: How diffie-hellman fails in practice. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15 (Denver, Colorado, USA, 2015), 5–17. ACM, New York, NY, USA. doi:10.1145/2810103.2813707. http://doi.acm.org/10.1145/2810103.2813707. (cited on page 26)

Aguillon, J. Ocaml ↔ Java Interface. https://github.com/Julow/ocaml-java. Accessed on April 29, 2019. (cited on page 69)

Alkassar, E.; Böhme, S.; Mehlhorn, K.; and Rizkallah, C., 2014. A framework for the verification of certifying computations. *Journal of Automated Reasoning*, 52, 3 (Mar 2014), 241–273. doi:10.1007/s10817-013-9289-2. https://doi.org/10.1007/s10817-013-9289-2. (cited on page 19)

Anand, A. and Rahli, V., 2014. Towards a formally verified proof assistant. In *Interactive Theorem Proving*, 27–44. Springer International Publishing, Cham. (cited on page 25)

Appel, A. W.; Michael, N.; Stump, A.; and Virga, R., 2003. A trustworthy proof checker. *Journal of Automated Reasoning*, 31, 3 (Nov 2003), 231–260. doi:10.1023/B:JARS.0000021013.61329.58. https://doi.org/10.1023/B:JARS.0000021013.61329.58. (cited on page 25)

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 pages 1 and 94)

Arkoudas, K. and Rinard, M. C., 2005. Deductive runtime certification. *Electr. Notes Theor. Comput. Sci.*, 113 (2005), 45–63. (cited on page 17)

Arrow, K. J., 1950. A difficulty in the concept of social welfare. *Journal of Political Economy*, 58, 4 (1950), 328–346. (cited on page 3)

Australian Electoral Commission, 2013. Letter to Mr Michael Cordover, LSS4883 Outcome of Internal Review of the Decision to Refuse your FOI Request no. LS4849. avaliable via http://www.aec.gov.au/information-access/foi/2014/files/ls4912-1.pdf, retrieved October 23, 2019. (cited on pages 2, 12, 16, and 94)

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 5)

Barras, B., 1996. Coq en coq. (1996). (cited on page 25)

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 58 and 61)

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 58)

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 84)

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 6)

Benaloh, J. and Tuinstra, D., 1994. Receipt-free secret-ballot elections (extended abstract). In *Proceedings of the Twenty-sixth Annual ACM Symposium on Theory of Computing*, STOC '94 (Montreal, Quebec, Canada, 1994), 544–553. ACM, New York, NY, USA. doi:10.1145/195058.195407. http://doi.acm.org/10.1145/195058.195407. (cited on pages 2 and 50)

Bernhard, M.; Benaloh, J.; Halderman, J. A.; Rivest, R. L.; Ryan, P. Y. A.; Stark, P. B.; Teague, V.; Vora, P. L.; and Wallach, D. S., 2017. Public evidence from secret ballots. In *Proc. E-Vote-ID 2017*, vol. 10615 of *Lecture Notes in Computer Science*, 84–109. Springer. (cited on pages 2, 15, 50, 57, and 83)

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 3, 21, and 35)

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 5)

Carr'e, B. A., 1971. An algebra for network routing problems. *IMA Journal of Applied Mathematics*, 7, 3 (1971), 273. (cited on page 44)

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

Chaum, D., 2004. Secret-ballot receipts: True voter-verifiable elections. *IEEE Security & Privacy*, 2, 1 (2004), 38–47. (cited on page 50)

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 60)

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

doi:10.1145/2815400.2815402. http://doi.acm.org/10.1145/2815400.2815402. (cited on page 13)

Chlipala, A., 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press. ISBN 0262026651, 9780262026659. (cited on page 35)

Cohen, E.; Dahlweid, M.; Hillebrand, M.; Leinenbach, D.; Moskal, M.; Santen, T.; Schulte, W.; and Tobies, S., 2009. Vcc: A practical system for verifying concurrent c. In *Theorem Proving in Higher Order Logics*, 23–42. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 19)

Constable, R. L.; Allen, S. F.; Bromley, H. M.; Cleaveland, W. R.; Cremer, J. F.; Harper, R. W.; Howe, D. J.; Knoblock, T. B.; Mendler, N. P.; Panangaden, P.; Sasaki, J. T.; and Smith, S. F., 1986. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. ISBN 0-13-451832-2. (cited on page 21)

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 15 and 16)

Coquand, T. and Huet, G., 1988. The calculus of constructions. *Inf. Comput.*, 76, 2-3 (Feb. 1988), 95–120. doi:10.1016/0890-5401(88)90005-3. http://dx.doi.org/10.1016/0890-5401(88)90005-3. (cited on page 22)

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 5)

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 5)

Cramer, R.; Damgård, I.; and Schoenmakers, B., 1994. Proofs of partial knowledge and simplified design of witness hiding protocols. In *Advances in Cryptology — CRYPTO '94*, 174–187. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 33)

Cramer, R.; Gennaro, R.; and Schoenmakers, B., 1997. A secure and optimally efficient multi-authority election scheme. In *Advances in Cryptology — EUROCRYPT '97*, 103–118. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 30)

de Bruijn, N. G., 1983. *AUTOMATH, a Language for Mathematics*, 159–200. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-642-81955-1. doi:10.1007/978-3-642-81955-1_11. https://doi.org/10.1007/978-3-642-81955-1_11. (cited on page 21)

de Moura, L.; Kong, S.; Avigad, J.; van Doorn, F.; and von Raumer, J., 2015. The lean theorem prover (system description). In *Automated Deduction - CADE-25*, 378–388. Springer International Publishing, Cham. (cited on page 21)

Delaune, S.; Kremer, S.; and Ryan, M., 2010a. Verifying privacy-type properties of electronic voting protocols: A taster. In *Towards Trustworthy Elections, New Directions in Electronic Voting*, vol. 6000 of *Lecture Notes in Computer Science*, 289–309. Springer. (cited on pages 2 and 50)

Delaune, S.; Kremer, S.; and Ryan, M., 2010b. *Verifying Privacy-Type Properties of Electronic Voting Protocols: A Taster*, 289–309. Springer Berlin Heidelberg, Berlin, Heidelberg. ISBN 978-3-642-12980-3. doi:10.1007/978-3-642-12980-3_18. https://doi.org/10.1007/978-3-642-12980-3_18. (cited on page 5)

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 5)

Diffie, W. and Hellman, M., 2006. New directions in cryptography. *IEEE Trans. Inf. Theor.*, 22, 6 (Sep. 2006), 644–654. doi:10.1109/TIT.1976.1055638. http://dx.doi.org/10.1109/TIT.1976.1055638. (cited on page 26)

Dijkstra, E. W., 1972. The humble programmer. *Commun. ACM*, 15, 10 (Oct. 1972), 859–866. doi:10.1145/355604.361591. http://doi.acm.org/10.1145/355604.361591. (cited on page 12)

Elections ACT, 2016. Electronic voting and counting. available via http://www.elections.act.gov.au/elections_and_voting/electronic_voting_and_counting, retrieved May 14, 2017. (cited on page 16)

ElGamal, T., 1985. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31, 4 (1985), 469–472. (cited on pages 26 and 28)

Erbsen, A.; Philipoom, J.; Gross, J.; Sloan, R.; and Chlipala, A., 2019. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, 1202–1219. doi:10.1109/SP.2019.00005. https://doi.org/10.1109/SP.2019.00005. (cited on page 13)

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 pages 1 and 94)

Firsov, D. and Uustalu, T., 2015. Dependently typed programming with finite sets. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming, WGP@ICFP 2015, Vancouver, BC, Canada, August 30, 2015*, 33–44. doi:10.1145/2808098.2808102. https://doi.org/10.1145/2808098.2808102. (cited on page 40)

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 5)

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 63)

Gentry, C., 2009. *A Fully Homomorphic Encryption Scheme*. Ph.D. thesis, Stanford, CA, USA. AAI3382729. (cited on page 29)

Geuvers, H.; Wiedijk, F.; and Zwanenburg, J., 2002. A constructive proof of the fundamental theorem of algebra without using the rationals. In *Types for Proofs and Programs*, 96–111. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 13)

Ghale, M. K.; Goré, R.; and Pattinson, D., 2017. A formally verified single transferable vote scheme with fractional values. In *Proc. E-Vote-ID 2017*, LNCS. Springer. This volume. (cited on pages 5 and 6)

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 5)

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 58)

Goldwasser, S.; Micali, S.; and Rackoff, C., 1985. The knowledge complexity of interactive proof-systems (extended abstract). In *STOC*, 291–304. ACM. (cited on pages 31 and 58)

Gonthier, G., 2008. The four colour theorem: Engineering of a formal proof. In *Computer Mathematics*, 333–333. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 13)

Gu, L.; Vaynberg, A.; Ford, B.; Shao, Z.; and Costanzo, D., 2011. Certikos: a certified kernel for secure cloud computing. In *APSys '11 Asia Pacific Workshop on Systems, Shanghai, China, July 11-12, 2011*, 3. doi:10.1145/2103799.2103803. https://doi.org/10.1145/2103799.2103803. (cited on page 13)

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, 12, and 94)

Hales, T. C.; Adams, M.; Bauer, G.; Dang, D. T.; Harrison, J.; Hoang, T. L.; Kaliszyk, C.; Magron, V.; McLaughlin, S.; Nguyen, T. T.; Nguyen, T. Q.; Nipkow, T.; Obua, S.; Pleso, J.; Rute, J.; Solovyev, A.; Ta, A. H. T.; Tran, T. N.; Trieu, D. T.; Urban, J.; Vu, K. K.; and Zumkeller, R. A formal proof of the kepler conjecture. *arXiv*. (cited on page 13)

Harrison, J., 1996. Hol light: A tutorial introduction. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, FMCAD '96, 265–269. Springer-Verlag, London, UK, UK. http://dl.acm.org/citation.cfm?id=646184.682934. (cited on page 21)

Harrison, J., 2006. Towards self-verification of HOL Light. In *Automated Reasoning*, 177–191. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 25)

Helios, 2016. The helios voting system. Http://heliosvoting.org/, accessed June 25, 2016. (cited on page 5)

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 58)

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)

Kaufmann, M. and Strother Moore, J., 1996. Acl2: an industrial strength version of nqthm. In *Proceedings of 11th Annual Conference on Computer Assurance. COMPASS '96*, 23–34. doi:10.1109/CMPASS.1996.507872. (cited on page 21)

Klein, G.; Elphinstone, K.; Heiser, G.; Andronick, J.; Cock, D.; Derrin, P.; Elkaduwe, D.; Engelhardt, K.; Kolanski, R.; Norrish, M.; Sewell, T.; Tuch, H.; and Winwood, S., 2009. sel4: formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, 207–220. doi:10.1145/1629575.1629596. https://doi.org/10.1145/1629575.1629596. (cited on page 13)

Kohno, T.; Stubblefield, A.; Rubin, A. D.; and Wallach, D. S., 2004. Analysis of an electronic voting system. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004*, 27–40. doi:10.1109/SECPRI.2004.1301313. (cited on pages 2 and 94)

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 5)

Kumar, R.; Myreen, M. O.; Norrish, M.; and Owens, S., 2014. Cakeml: a verified implementation of ML. In *Proc. POPL 2014*, 179–192. ACM. (cited on pages 13 and 55)

Küsters, R.; Truderung, T.; and Vogt, A., 2011. Verifiability, privacy, and coercion-resistance: New insights from a case study. In *2011 IEEE Symposium on Security and Privacy*, 538–553. doi:10.1109/SP.2011.21. (cited on pages 2 and 50)

Landin, P. J., 1964. The mechanical evaluation of expressions. *The Computer Journal*, 6, 4 (1964), 308. (cited on page 52)

Leroy, X., 2006. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, 42–54. doi:10.1145/1111037.1111042. https://doi.org/10.1145/1111037.1111042. (cited on page 13)

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 69)

Lewis, S. J.; Pereira, O.; and Teague, V. Ceci n'est pas une preuve. https://people.eng.unimelb.edu.au/vjteague/UniversalVerifiabilitySwissPost.pdf. Accessed on October 17, 2019. (cited on pages 1 and 94)

Locher, P. and Haenni, R., 2014. A lightweight implementation of a shuffle proof for electronic voting systems. In *44. Jahrestagung der Gesellschaft für Informatik, Informatik 2014, Big Data - Komplexität meistern, 22.-26. September 2014 in Stuttgart, Deutschland*, 1391–1400. https://dl.gi.de/20.500.12116/2747. (cited on pages 4 and 69)

McConnell, R.; Mehlhorn, K.; Näher, S.; and Schweitzer, P., 2011. Certifying algorithms. *Computer Science Review*, 5, 2 (2011), 119 – 161. doi:https://doi.org/10.1016/j.cosrev.2010.09.009. http://www.sciencedirect.com/science/article/pii/S1574013710000560. (cited on page 17)

Mehlhorn, K. and Näher, S., 1995. Leda: A platform for combinatorial and geometric computing. *Commun. ACM*, 38, 1 (Jan. 1995), 96–102. doi:10.1145/204865.204889. http://doi.acm.org/10.1145/204865.204889. (cited on page 19)

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)

Menezes, A. J.; Vanstone, S. A.; and Oorschot, P. C. V., 1996. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edn. ISBN 0849385237. (cited on page 35)

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)

Milner, R., 1972. Implementation and applications of scott's logic for computable functions. *SIGPLAN Not.*, 7, 1 (Jan. 1972), 1–6. doi:10.1145/942578.807067. http://doi.acm.org/10.1145/942578.807067. (cited on page 21)

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 14)

Myreen, M. O. and Davis, J., 2014. The reflective milawa theorem prover is sound - (down to the machine code that runs it). In *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, 421–436. doi:10.1007/978-3-319-08970-6_27. https://doi.org/10.1007/978-3-319-08970-6_27. (cited on page 13)

Nipkow, T.; Paulson, L. C.; and Wenzel, M., 2002a. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, vol. 2283 of *Lect. Notes in Comp. Sci.* Springer. (cited on page 21)

Nipkow, T.; Wenzel, M.; and Paulson, L. C., 2002b. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, Berlin, Heidelberg. ISBN 3-540-43376-7. (cited on page 19)

Norell, U., 2009. Dependently typed programming in agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming*, AFP'08 (Heijen, The Netherlands, 2009), 230–266. Springer-Verlag, Berlin, Heidelberg. http://dl.acm.org/citation.cfm?id=1813347.1813352. (cited on page 21)

O'Neill, O., 2002. *A Question of Trust*. Cambridge University Press. (cited on page 15)

Otten, J., 2003. Fuller disclosure than intended. (2003). http://www.votingmatters.org.uk/ISSUE17/I17P2.PDF. Accessed on October 17, 2019. (cited on pages 6, 55, and 57)

Owre, S.; Rushby, J. M.; and Shankar, N., 1992. Pvs: A prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*, CADE-11, 748–752. Springer-Verlag, London, UK, UK. http://dl.acm.org/citation.cfm?id=648230.752639. (cited on page 21)

Paar, C. and Pelzl, J., 2009. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer Publishing Company, Incorporated, 1st edn. ISBN 3642041000, 9783642041006. (cited on page 35)

Pattinson, D. and Schürmann, C., 2015. Vote counting as mathematical proof. In *Proc. AI 2015*, vol. 9457 of *Lecture Notes in Computer Science*, 464–475. Springer. (cited on pages 5 and 17)

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 5)

Paulin-Mohring, C., 1993. Inductive definitions in the system coq - rules and properties. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, TLCA '93, 328–345. Springer-Verlag, London, UK, UK. http://dl.acm.org/citation.cfm?id=645891.671440. (cited on page 22)

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 61)

Pollack, R., 1998. How to believe a machine-checked proof. *Twenty Five Years of Constructive Type Theory*, 36 (1998), 205. (cited on page 25)

Rivest, R. L., 2008. On the notion of software independence in voting systems. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366, 1881 (2008), 3759–3767. https://royalsocietypublishing.org/doi/10.1098/rsta.2008.0149. (cited on page 15)

Rivest, R. L.; Adleman, L.; Dertouzos, M. L.; et al., 1978. On data banks and privacy homomorphisms. (1978). (cited on page 29)

Rivest, R. L. and Shen, E., 2010. An optimal single-winner preferential voting system based on game theory. In *Proc. COMSOC 2010*. Duesseldorf University Press. (cited on page 38)

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 5)

Schneier, B., 1995. *Applied Cryptography (2Nd Ed.): Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., New York, NY, USA. ISBN 0-471-11709-9. (cited on page 35)

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 3, 38, and 47)

Schürmann, C., 2009. Electronic elections: Trust through engineering. In *Proc. RE-VOTE 2009*, 38–46. IEEE Computer Society. (cited on page 17)

Slind, K. and Norrish, M., 2008. A brief overview of hol4. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '08 (Montreal, P.Q., Canada, 2008), 28–32. Springer-Verlag, Berlin, Heidelberg. doi:10.1007/978-3-540-71067-7_6. http://dx.doi.org/10.1007/978-3-540-71067-7_6. (cited on page 21)

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 42)

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 17)

Tarski, A., 1955. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5, 2 (1955), 285—309. (cited on page 42)

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 60)

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 5)

Vogl, F., 2012. *Waging War on Corruption: Inside the Movement Fighting the Abuse of Power*. Rowman & Littlefield. (cited on page 15)

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 page 61)

Wilcox, J. R.; Woos, D.; Panchekha, P.; Tatlock, Z.; Wang, X.; Ernst, M. D.; and Anderson, T. E., 2015. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation,*

Portland, OR, USA, June 15-17, 2015, 357–368. doi:10.1145/2737924.2737958. https://doi.org/10.1145/2737924.2737958. (cited on page 13)

Wolchok, S.; Wustrow, E.; Halderman, J. A.; Prasad, H. K.; Kankipati, A.; Sakhamuri, S. K.; Yagati, V.; and Gonggrijp, R., 2010. Security analysis of india's electronic voting machines. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10 (Chicago, Illinois, USA, 2010), 1–14. ACM, New York, NY, USA. doi:10.1145/1866307.1866309. http://doi.acm.org/10.1145/1866307.1866309. (cited on pages 2, 11, and 94)

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)

Zhao, J. and Zdancewic, S., 2012. Mechanized verification of computing dominators for formalizing compilers. In *Certified Programs and Proofs*, 27–42. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 13)