Applied Formal Methods for Elections

Jian Wang

Advisor: Carsten Schürmann
Submitted: March 2016

IT University
of Copenhagen

Abstract

Information technology is changing the way elections are organized: In many countries voters are nowadays voting with electronic voting machines, some countries even offer internet elections as alternative voting channels, and it is to be expected that information technology will change the way ballots are tabulated and parliaments are elected. There are many reasons explaining why election commissions around the world pursue an agenda of digitizing elections. Indeed, technology renders the electoral process more efficient, but things could also go wrong: Voting software is complex, it consists of thousands of lines of code, which makes it error-prone. Software and hardware problems may cause delays at polling stations, or even delay the announcement of the final result. This thesis describes a set of methods to be used, for example, by system developers, administrators, or decision makers to examine election technologies, social choice algorithms, and voter experience.

Technology: Verifiability refers to voting software producing evidence so that can be retroactively analyzed, in case that something went wrong, or there is a suspicious of foul play. Although verifiability is an elegant concept, it is by no means easy to judge, if an implementation implements verifiability correctly. There are two ways to analyze implementations for verifiability, first statically, i.e. before the technology is deployed, which gives the developers the opportunity to fix issues during development time, or second dynamically, i.e. monitoring while an implementation is used during an election, or after the election is over, for forensic analysis.

This thesis contains two chapters on this subject: the chapter *Analyzing Implementations of Election Technologies* describes a technique for checking verifiability properties directly on the source code of e-voting systems using modern code scanning tools. Verifiability properties are expressed in linear temporal logic, then translated into Büchi automata and checked using off the shelf static analyzers that compute all possible execution paths. We demonstrate the technique using a particular took, the Coverity static analyzer. In the chapter *Epistemic Policies for Voting Systems*, we turn our attention to the logs generated by e-voting systems for policy compliance checking. The novel contribution is that we use epistemic linear time logic to express properties about distributed e-voting systems that then be checked by model-checkers.

Algorithms: The advances in technology open the possibility that the very algorithms and social choice functions we use in elections are going to evolve. Besides first-past-the-post, there are many variants of D'Hondt, Sainte-Laguë and single transferable vote (STV), and these algorithms continue to be adjusted, evolved, and refined.

This thesis contains a chapter *Verifying Voting Schemes* that focuses on the use of formal methods to ensure that these algorithms have the intended meaning and conform to the desired democratic principles. As a case study, we define two semantic criteria for STV schemes, formulated in first-order logic over the theories of arrays and integers, and show how bounded model-checking and satisfiability modulo theories (SMT) solvers can be used to check whether these criteria are met.

Voter Experience: Technology profoundly affects the voter experience. These effects need to be measured and the data should be used to make decisions regarding the implementation of the electoral process.

This thesis contains a chapter *Measuring Voter Lines* that describes an automated data collection methods for measuring voters' arrival and waiting time, and discusses statistical models designed to provide an understanding of the voter behavior in polling stations.

Acknowledgements

First of all, I want to thank my supervisor, Carsten Schürmann for guiding me through my PhD study, for leading me to an amazing research area – theoretical computer science. He could always find time to meet me and discuss about my work with great patience and enlightenment. Without him, I could not image of being able to finish my study and thesis.

I also would like to thank all my friends and colleagues from PLS group and DemTech project, Daniel, Marco, Lorena, both Nicolas, Agata, Peter, Alec, etc., some of whom inspired me on my research, and some of whom arranged wonderful after work events. I've enjoyed my work and life here at ITU for the past years because of them.

I would like to thank my wife - Ruiyu Lin, my parents and parents-in-law, for their constant support. I feel so lucky with you all standing behind me and supporting me all the time.

I also want to thank Prof. David Basin from ETH Zürich for having me as a visiting student, and whose work on security policy checking inspired my PhD research topic.

In addition, I would like to thank Nicole Bridges, who helped me with my English writing.

Contents

Chapter 1

Introduction

An election is a decision making process that allows voters choose to express their preferences via ballots. Elections enable voters to participate in the democratic process in a representative democracy. Examples for elections include parliamentary elections, local elections, association elections, private organization elections and so on. The *electoral cycle* [78, 73] is a visual planning and training tool for assisting the organization of elections. The electoral cycle is divided in three periods, and each includes many processes,

- the pre-electoral period, including planning, training, voter registration, etc.
- the electoral period, including vote casting, ballot counting, result verification, etc.
- the post-electoral period, including audits and evaluations, legal reform, etc.

The aim of this thesis is to show that formal methods can be used to check and to improve the application of election technology and the voter experience in polling places. In this thesis, we use formal methods to analyze the properties of two processes in the electoral period - *vote casting* and *ballot counting*.

Vote Casting There are various vote casting forms, including paper voting, mail voting and electronic voting (e-voting). Voters can either cast their ballots in polling stations (via e-voting, paper voting) or remotely (via internet e-voting, mail voting). Information technology is changing the way elections are organized: as a modern ballot casting method, e-voting has gained popularity alongside the rapid prevalence of computers and smartphones. In some countries [51] voters are nowadays voting with e-voting machines in polling places such as the direct-recording electronic (DRE) systems in the US and the vVote system in Victorian State (Australia) election. Voters can even vote remotely in a few countries, such as Estonia and Norway. E-voting systems can be very complicated, and they usually run on even more complicated operating systems, both of which can easily have glitches and bugs, which could be exploited by attackers and adversely affect the election quality. Besides the vote casting forms, technology also profoundly affects the voter experience in polling stations. These effects also need to be measured and the data should be used to make decisions regarding the implementation of the electoral process.

Ballot Counting Proportional representation and plurality voting are the two major methods for allocating seats in parliamentary elections. In proportional representation electoral systems, divisions in an electorate are reflected proportionately by the application of *voting schemes* (ballot counting methods). There are many voting schemes, which turn ballots into decisions of distributing mandates seats, used around the world. The ballot content for a proportional representation system can be either a single choice or a preferential rank of multiple choices of candidates. Seats distribution computing methods (voting schemes) for single choice voting include the largest remainder method like Hare quota and the highest average method like D'Hondt method. Voting schemes for ranked voting include the instant-runoff voting, Borda count, single transferable vote (STV) and so on.

In this thesis, we use formal methods to analyze the electoral cycles, in particular the vote casting process and ballot counting process. We focus on three topics, *properties of election technology*, *criteria of voting schemes* and *user experience in polling stations.*

1.1 Properties and Model Checking

We make use of time and temporal descriptions in election requirement, such as "the election should be held in two weeks after the prime minister declared it", "a voter's ID should be checked before he/she is allowed to vote", "a voter can not vote again after he/she has voted". We use words like "after", "in two weeks" to show that something will or should happen in the future, and we use "before" to describe things in the past. There is a type of formal logic, *temporal logic*, which is designed to represent and reason about time and temporal information.

As an extension of classical logic, temporal logic includes the operators, such as the conjunction ($\wedge$) and disjunction ($\vee$) of two formulas, the negation ($\neg$) of a formula. In addition, temporal logic enriches classical logic by the temporal operators like $\bigcirc$, $\Diamond$, $\Box$, which represents "at next state in time", "at some future state in time" and "at all future (include current) states in time" respectively. For example, when it is required that "whenever a send message action happens, then some time later there should be a receive message action take place.", the formula $\Box(send\_message \rightarrow \Diamond receive\_message)$ describes this policy. These temporal properties usually can be a safety property, which states that nothing should go wrong, or a liveness property, which states that the right thing should finally happen. [2]

The notion of security policy and security model was introduced in 1982 by Goguen and Mesequer [66]. We call these requirements of system states system properties (or security policies). Note that there is a subtle different between system properties and security policies in some scenarios, however, we consider them interchangeable in this thesis. Temporal logic can be used to express the requirements of systems states, and it plays an important role in system verification.

In this thesis we will use temporal logic to describe system properties, and use model checking method to verify them. Model checking started from the early works [37, 101] and has been a widely used method for system property verification. Model checking refers to exhaustively and automatically checking if the model meets a specification, in our case the temporal formulas. For checking some properties, the model checker continues until all possible states are checked. A good overview of the model checking can be found [9].

Instead of enumerating reachable states in model checking, symbolic model checking considers a large set of possible states at one time. Symbolic model checking with Binary Decision Diagrams (BDDs) has been successfully used for formal verification of finite state machine, but BDDs may grow exponentially for the verification. To solve this problem, a method called Bounded Model Checking (BMC) is introduced by Biere [23]. It was an attempt to replace BDDs with SAT and SMT solvers in symbolic model checking, and to find counter examples with a SAT-solver.

1.2 Election Technology

1.2.1 Code Checking

Verifiability is a key feature of e-voting technology, as it allows voters and auditors using evidence, such as a ballot receipt, to verify the correctness of the result. Verifiability entails that a voter or administrator can inspect evidence that is generated during the run-time of the election technology, and judge, beyond some level of doubt, that the process is correct. The concept of *end-to-end verifiability* [83, 82, 33] has emerged as a valuable mechanism for accountability of e-voting technology, and it includes three properties, cast as intended, stored as cast and counted as stored. For each of these properties, it is time intensive to check for implementations, and those properties for end-to-end verifiable systems normally are only checked in system design level. Although there has been a considerable amount of work on devising and checking verification procedures for voting systems on the design specification (see related work section), there is relate little work has been done in the fine-tuned checking in software implementation of election technologies. In Chapter 2, we are exploiting the possibility of applying software model checking to election technologies by using off-the-shelf static code analysis tools to verify whether an e-voting implementation actually produces evidence correctly for the verification procedure.

In our application, we focus on the code scanning based method for evidence flow analysis. Evidence flow is the flow of generating the evidence from starting point to an ending point in the middle of ballot processing. Taint analysis is an important branch for data flow analysis, which can be used to track the evidence flow. It can trace the data flow throughout the program paths through symbolic execution, which provide an important tool for evidence checking in voting systems, because the interaction between the server (voting center) and the client (voting machine).

We show that it is possible to carry out the checks in two modes. Mode 1, we can track the forward flow of the ballot from input, to the place in the source code where the evidence such as a ballot receipt for later checking is generated. By mode 1 checking, we are able to verify that the evidence is generated correctly. And Mode 2 is the backward flow analysis, in which we can check that every evidence generated in the end should correspond to some input, which means the evidence should be able to be traced back to the input source. For example, a ballot receipt should be able to trace back to to a starting point, where the ballot is cast.

In the method we use, a Büchi automaton is built from a property expressed by temporal logic, which is then checked with the implementation's function call paths generated by flattening function calls following the idea from taint analysis. We prove that our abstraction method is a TI abstraction, which means we can replace the source code property checking problem by checking the property in the function call path. *TI abstraction* [62], as defined in Definition 1.2.1, can be used to achieve a fast check of the complex systems in a high level description of the source code, and yet be able to check the properties for the original model.

Definition 1.2.1 (TI Abstraction)**.** f is TI abstraction iff, for all formulas p if p in $Th(S)$ then $f(p)$ in $Th(f(S))$, where f is an abstraction between a pair of formal systems. $Th(S)$ is the theorems in formal system S.

1.2.2 Log Checking

A log refers to a document that records relevant events in the order they occurred. The purpose of logs are numerous, including retroactive forensic analysis in the case that something went wrong, providing evidence that assigned tasks have been conducted responsibly, and conducting statistical analysis about the system operation. A poll book is a classic example, where the distribution of ballots is carefully recorded and used to estimate voter participation at the end of voting day. But this is not everything. There is an extra feature of these kind of logs that is worth highlighting: Complete logs can create trust in those who read them, in particular because they often allow the reader to reconstruct the steps in between and therefore retrace all steps of a process and to gain confidence in whatever the process set out to execute.

Besides the traditional usage, logging is also highly recommended in software system engineering security standards [100]. There are many logging standards or guides created for a variety of systems. These standards define the log structure and log content, like the NIST's "Guide to Computer Security Log Management" [84] and the "NCSA Common log format" [76]. "Secure programming with static analysis" [28] advocates some basic requirements for good logging: time-stamped log entries, consistent logging of all important actions, and controlled access to log data. Without the time-stamp, the forensic value will be much less important, since time-stamps can help to reconstruct the sequence of logged events.

When we turn our attention to logging systems that are prevalently used in computer systems, it is clear that those kind of logs aim to achieve very similar outcomes. Web-server access logs record who connected to the server and from which IP address, and in the case of a break-in (that has to be noticed by other means like intrusion detection system), often provides some evidence revealed by forensic analysis. The messages log file in a Linux system, for example, contains information about which process did what, and if a problem was detected or an exception was raised. Finally, logs can serve to provide statistical information, as how many connection requests were received or how many pages were served. There is one thing, however, that distinguishes computer logs from traditional logs. They do not aim to be as complete as the traditional book keeping, because there are simply too many steps that could be logged. Thus nobody would expect a log to generate trust into the correctness of an entire application.

When a system fails during run time due to some errors, these errors are reproducible with enough information logged. By following the log entries, a system administrator can reconstruct the sequence of events leading to the errors. In addition, logs can be used for analysis like collecting data for voting sessions and voters [69]. The role of logging and logs for electronic voting systems, which is a necessary step towards a trustworthy e-voting system, is presented and studied in Chapter 3.

Though log checking is different from the dynamic execution states checking of a system, it is still possible to track some information from the system execution with proper deployment of logging and log monitoring components, and it can also be checked dynamically at run-time. Log handling always starts with installation of a logging framework in a system, which records system events log entries during the operation of the system. After or during recording, logs can be used for checking adherence of system properties with model checking methodology.

A lack of information in the logs may lead to the negligence of some problems, as some properties may not be able to check due to missing information. Too few entries will lead to the omission of some important information, and when doing automatic log analysis, it slows the log checking program's efficiency. Thus a certain amount of data from the logs is always required before running the aforementioned checking and analysis. So some systems log as much information as possible. In scenarios like e-voting, extensive logging may adversely affect the confidentiality and privacy of the voter. However, the log information leakage analysis is not covered in this thesis.

1.3 Voting Schemes

Voting schemes in elections are functions that compute the seat distribution from a set of preferences recorded on ballots. In the study of social choice functions, various new voting systems are created and pre-existing schemes are adjusted and refined constantly. Voting schemes are designed to follow certain social choice criteria like the Condorcet criterion [1], defined as "the candidate who wins against each other candidate in a one-on-one contest should be the election winner". Voting schemes can be analyzed according to many different criteria. For example, the Condorcet criterion is satisfied by the *Schulze* and *Copeland* voting schemes, but not the *single transferable vote* (STV) and its variants.

However, no voting scheme exists that would satisfy all reasonable general criteria simultaneously. In the case of preferential voting, Arrow's impossibility theorem [6] states that no scheme can be designed to satisfy all of the three fairness criteria simultaneously:

Unanimity. If all voters prefer candidate A over candidate B, then A is ranked over B in the election result.

Independence of irrelevant options. If some voters change their ballot but keep the relative position of candidates A and B on their ballot, then the relative position of A and B remains unchanged in the election result.

Non-dictatorship. There is no single voter whose preferences always prevail in the election result.

The possibility to use computers for counting ballots allows us to design new voting schemes that are arguably fairer than existing schemes designed for hand-counting. In Chapter 4, we study the applicability of formal methods for the purpose of criteria checking of voting schemes. We argue that formal methods can and should be used to ensure that such schemes behave as intended and conform to the desired democratic properties. Specifically as examples, we define two semantic criteria for STV schemes, which are used in the Victoria State election and many others, and formulate them in first-order logic over the theories of arrays and integers, and show how bounded model-checking and SMT solvers can be used to check whether these criteria are met. As a case study, we then analyze an existing voting

[1]`https://en.wikipedia.org/wiki/Condorcet_criterion`

scheme for electing the board of trustees for a major international conference and discuss its deficiencies. We also show in Chapter 4 that the variant of STV implements the majority rule instead of the proportional rule as intended, which violates the original design intention of the social choice algorithm.

1.4 Voter Experience

When talking about voter experience, we are referring to the impact of choices that the election officials make on the voter during the electoral cycle. These includes for example, the voter registration, the location and design of polling places, the management of voter flow, voting technology, etc. [16]

These effects need to be measured and quantified, and the insight gained can be used to help organizing the elections. For example, the voters may balk if there is a long queue ahead, even though the queue may actually move fast, which will affect the quality of the voter experience. Shortening the queues in polling stations and providing the expected waiting time for the voters are two possible solutions of the balk-out problem. In Chapter 5 of the thesis, we will look at the voter experience of an election, more specifically, queues in the polling stations.

In Chapter 5, we present an automated data collection technique called *white boxes*-technique, which is designed

- to analyze the voter behavior in polling stations including the measurement of arrival and waiting times and the determination of arrival frequency;
- to assist the management of polling places to make decisions regarding the distribution of resources and to identify areas for future improvement;
- and to provide hard data to guide the political decision making process with respect to the choice of voting technologies.

We describe the technology that we used, analyze its security, give an empirical analysis how the technique compares to traditional manual data collection techniques, such as the ones proposed for US elections [2] by the

[2]`http://web.mit.edu/vtp/`

CalTech/MIT Voting project, using data collected during the 2015 Danish parliamentary election. We record the voters‘ arrival and departure times and analyze the average time in five polling places in the context of queueing theory [41].

1.5 Related Work

This section gives a general survey of the related work with respect to different topics mentioned above.

Properties and Model Checking Policy languages are usually tailored to their application domains. There are languages that capture time and time-outs [13], access control [60], and aggregation [12]. Much related work has been done in the field of security policy formalization and security policy checking. Perhaps most closely related to our work is the work of Basin that covers metric first-order temporal logic (MFOTL) [14], which supports security policy that refer to "real-time", and its generalization to aggregates [12]. The application of epistemic temporal logic to security information flow analysis is discussed in [10], and there has also been some work regarding the aximatization of epistemic temporal logic [20]. In recent years, information-flow security properties were studied in systems as HyperLTL and HyperCTL* [38] express information-flow policies by explicit quantification over multiple traces.

An introduction to temporal logic and its application can be found in [55]. [88, 90, 55] show examples how temporal logic can be used in program specification, and [9, 52, 74] show that temporal logic can be used for system verification. Some application of temporal logic in multi-agent systems are discussed in [56, 7].

As to the translation from LTL to Büchi Automata, we use the LTL2BA program created by Gastin [61] in Chapter 2. There are some other translation methods, such as [74, 47, 107, 105, 8]. Holzmann [74] in the model checker Spin also introduced a translation method, and LTL2BA uses the same modeling language as SPIN. [65] also shows a method of model checking for programming languages. More algorithms for the translation are listed on the SPOT wiki [3].

An introduction to bounded model checking using SAT solving is provided in [36, 5]. [79] discusses a methodology for formal analysis of software

[3]`http://spot.lip6.fr/wiki/LtlTranslationAlgorithms`

programs via SAT-based bounded model checking. More discussion about SAT based bounded model checking is given in [59, 23, 22]

[34] presents the idea of using finite automaton to check securities for piece of source code. It uses a pushdown automaton to represent the program, and a finite automaton for the policy. Our method is similar to this one but we formalize the properties and the generating of the paths. [94] introduces a tool to find some security problems in file reader socket Java code, like resource injection, path manipulation. Another useful tool for Java project analysis is the Java path finder [113], which requires extra annotation to the source code to run non-functional property checking. A list of source code analysis tools are listed in OWASP website [4].

Logging and Log Analysis Much work has been done in the area of logging and log analysis. We focus only on some relevant related work. The logging topic has been studied in the aspects like replayable logs [42], the integrity of logs [43], and usage of logs for process mining [112]. The model-checking based log auditing is discussed in [103], which introduces the idea of using temporal logic model checking for logs.

AccuVote Optical Scan (AV-OS) terminal event log analysis derived from an abstract finite state model is presented in [4]. In “Automated Analysis of Election Audit Logs” [17], a method to analyze logs from Direct Recording Electronic (DRE) voting machines is discussed, in which a web application and toolset for uploading and analyzing DRE voting logs and ballot image files are built. As a continued work from [4], “A Systematic Approach to Analyzing Voting Terminal Event Logs” [96] introduces a forensics tool based on context-free grammars, which also focuses on the AV-OS terminal logs. Voting logs for Estonia’s I-Voting analysis is presented in [69] and [70].

E-Voting and Verification E-voting systems’ report and analysis can be found in [68, 71] for I-Voting system Estonian elections, for the Norwegian system [64, 63], for the Australian vVote project [31, 45, 44, 46] and others [35, 106]. The analysis of the remote e-voting systems is discussed in [87], and a world map that depicts the world wide use of e-voting and new voting technologies is provided by E-voting.CC [51].

There is some previous work on the formal analysis of voting schemes using methods and tools from the computer aided verification and auto-

[4]`https://www.owasp.org/index.php/Source_Code_Analysis_Tools`

mated deduction communities in our sense, such as the formal analysis of actual implementations of such schemes [92, 85, 39]. Methods regarding E-voting system verification can be found in [85, 40, 111, 86], most of which focus on the system design level. In McGaley's thesis [93], the ballot secrecy and the voter verified traits are discussed, and also in Aida's thesis [1], the cryptographic techniques for e-voting are presented.

Voting Schemes Voting schemes have been investigated by social choice theorists for many decades. These tend to be mathematical analyses which prove various (relative) properties of different voting schemes: see [98, 6]. Such work tends to concentrate on what we have referred to as theoretical schemes and is often couched in terms of a formal theorem and its proof in natural language.

Many general criteria that voting schemes preferably should satisfy have been proposed in [26]. There is also a significant body of research on various properties of vote-casting schemes, particular security properties [110, 67, 102].

Voting Lines and Queueing Data Collection The MIT/CalTech project [109] has conducted research in the general area of voter line formation and queuing behavior. A study conducted during the 2008 US presidential primary election compares the formation of lines in polling stations that use DRE voting machines to those using non-DRE voting systems by analyzing manually collected data [108]. An alternative way to study voter lines is by a post-election survey involving local election officials as documented in [3]. [16] provides the report and recommendations on the voter experience of American presidential elections, in which the long queue problem is also discussed.

With respect to the automatic collection of queueing data, [30] describes a smartphone app for collecting waiting times via crowdsourcing information of client side actively uploading location information from GPS and Wi-Fi to a server. This technique requires active participation of those waiting, which distinguishes it from our study. A related way of monitoring waiting times for public transit is provided in [116] predicting bus arrival times by data analysis of cell tower signals and audio recordings, and similar work has been done in [54].

1.6 Structure and Contribution

This thesis is a collection of manuscripts and published papers. It consists of three parts, which covers the three aforementioned topics, election technology, voting schemes and voter experience.

The first part contains two chapters, Chapter 2 and Chapter 3, which discuss the verifying properties of e-voting system technologies through code scanning and log checking respectively. Chapter 2 is a joint work with Carsten Schürmann, and Chapter 3 is a joint work with Carsten Schürmann and Daniel Gustafsson. For both chapters, I contributed to the theory and the experiments for code checking and log checking.

The second part is Chapter 4, and it focuses on voting scheme criteria and their verification. It is a paper published in Journal of Information Security and Applications (JISA) 2014, and it is a joint work with Bernhard Beckert, Rajeev Goré, Carsten Schürmann and Thorsten Bormer. Bormer and I were in charge of the SMT solver and bounded model checking part, in addition I added some extra examples of the property formulating.

The last part is Chapter 5, which talks about the voter experience. It provides a way of collecting voter behavior data, voters' arrival and departure time, for the voter experience analysis. This paper will appear in International Conference for E-Democracy and Open Government (CeDEM) 2016, and it is a joint work with Carsten Schürmann. I mainly contributed to the design and implementation of the white box-technique experiments, and I also processed and analyzed the collected data. In addition, I took part in the manual data collection.

The tools and implementations mentioned in this thesis can be found on `http://itu.dk/people/jwan/`.

1.7 Conclusion and Future Work

This thesis aims on improving the current election quality, and it provides and demonstrates possible methods to do so from both technical and theoretical point of views. With all the work discussed in this thesis, we reach a conclusion that election procedure can be checked and improved with applied formal methods. We tested and showed that there are ways to make the election and e-voting an even better way to serve democracy by making the e-voting move towards verifiability and making the experience in polling places better.

The future work of this thesis can be carried out in the following directions with respect to each chapter. First, the code scanning chapter can be continued by extending from just function names to variables and data flows after we obtain the license to do taint analysis. and also can be continued by dealing with the complexity problem. When we test our method in the Victorian vVote system, complexity is not considered since the system is relatively small, but efficiency needs to be considered when we want to make it universally applicable. For the log checking chapter, the long term goal is to formalize end to end verifiability, as described in Section 1.2, with log checking. For the voting schemes part, the future work is to improve the reach and the efficiency of SMT-based analysis as described in Section 4.4. This will allow us to investigate larger classes of voting schemes and to use more complex criteria. We also plan to extend our analysis to criteria that measure the quality of election results based on difference measures [95] in addition to yes/no criteria. The future work for the white box chapter is the automatic online data collection, which means we need to solve the problem of a reliable secure way of transferring the data to a server that offers online real time information of the polling stations. Besides these chapters, the continued work of this thesis can also be the analysis of processes in the electoral cycle not covered, such as the voter registration process and the result verification process.

Chapter 2

Analyzing Implementations of Election Technologies[1]

2.1 Introduction

Software verification is a difficult endeavor. Given a program and a specification, we try to argue, that when executed, the program behaves as specified. Even with all the progress in the field, software verification of production quality software is still an elusive goal. There are several reasons for this. (1) Production quality software systems are complex, they use different programming languages, build on large bodies of libraries, and require configurations. (2) The specification of a program can be expressed in a logic; there are many logics that require different levels of support. (3) It is possible to find specifications for small programs, but this task becomes increasingly difficult the larger and the more complex a program becomes. (4) Software verification is extremely time intensive, and verified software proofs are often bridle to changes of the program that is being checked. Small changes in an implementation may require large parts of the system to be reverified.

For election technologies, a trend has emerged to require that voting software, used for example in kiosk based voting machines or internet voting solutions, is verifiable. Verifiability entails that a judge (or in many cases the voter) can inspect evidence that is generated during the execution (run-time) of the election technology and judge, beyond some level of doubt, that the result is correct. Examples of such evidence include cryptographic proofs, such as proofs of knowledge, logical proofs, such as

[1]Joint work with Carsten Schürmann

formal certificates and log files, and statistical proofs, for example through auditing paper evidence (which are less important for this chapter). The advantage of evidence based computation is that it will be (at least in theory) checkable by others, using their own evidence checking tools, such as zero-knowledge proof checkers, certificate checkers, or log analyzers. Trust in election technologies therefore does not rely solely on the correctness proof of an implementation (that is hard to come by as election technology solutions are huge), but is instead diversified by offering control mechanisms for post-election analysis.

This means that instead of full-fledged analysis using software verification (which would provide the highest levels of quality assurance, but at enormous cost), we may, at least in this very specific settings where software systems run only for limited amount of time, conduct meaningful and more lightweight analyses on the source code by checking that the flow of evidence is correct. Evidence flow analysis combined with the post-mortem check on the correctness of the evidence, yields a much more appealing argument for the correctness and thus also trustworthiness of an election technology than software verification alone ever could.

Contributions: This chapter makes the following contributions.

- Evidence analysis. We use LTL as a policy language to express the property of evidence flow. Here, we assume that there is a set of functions used to generate evidence. Evidence analysis consists of two parts. First, we check that the designated sources of information to compute evidence flow into the designated function that computes the evidence, and second we check that the evidence generated flows to the designated sinks, for example, a bulletin board, a SMS delivery service, or a log file.

- We use standard code scanning technologies, in our case Coverity, but any other customizable code scanning product would work, to compute the set of all abstract execution paths. The advantage of using such tools is that they work with the real programming language, and not just a toy subset of it. We work only with TI abstractions [62], taking advantage of the fact that if the abstract version of the execution path violates the policy, then the real execution path also violates it. We use standard LTL model checking techniques to validate abstract execution path.

- We apply the proposed technique to a real world example, the vVote system [2], an end-to-end voter verifiable kiosk based voting system that was used in the state elections of the state of Victoria in Australia.

2.2 Evidence Flow

Evidence in this chapter refers to the proof that the e-voting system implementation processes the ballot correctly, and flow means a starting point (source) and an ending point (sink) exist for the processing of the ballot. As required by end-to-end verifiability, the ballot should be cast as the voter intended, recorded in the server as the voter cast, and counted as recorded. There are corresponding evidences for each of these three verifiability properties. For example, the SMS text return message proves that the ballot has been received by the service correctly, and a ballot receipt generated after the voter has voted in a kiosk voting machine provides an evidence for checking the ballot be correctly counted. It's also possible to composite evidence flow, so the source and sink flow can be plugged together. For example, once the source and sink evidence flow is checked for an e-voting server for different procedures, it's possible to claim some properties for the whole system, such as that the end-to-end verifiability contains three individual requirements.

We propose two modes to check the evidence flow. Mode 1 tracks the forward flow of the ballot from initial input, until the generating of the final evidence such as the SMS return code or a ballot receipt. Through mode 1 checking, we are able to verify that the evidence has been generated correctly following the e-voting system's procedures, such as ballot validation, ballot storing, receipt generation, etc. Mode 2 is the backward flow analysis, in which we want to show that evidence generated in the end should correspond to a certain input, which means the evidence should be able to be traced back to the input of a ballot. For example, a ballot receipt should correspond to a starting point, where the ballot is entered.

There are different flavors of evidence flow in the e-voting implementation, such as function calls, data flows and class hierarchy. In this chapter, we assume all critical handling of the evidence, such as ballot validation, happens in functions, therefore we use the path of function calls to verify the evidence flow. However, our method is not restricted to functions, and it can be extended with variables for data flow analysis, which will be covered in future work.

[2] `https://bitbucket.org/vvote/`

2.3 Policy Language

When the e-voting system is designed, the behaviors of the system are predefined in the specification. Such behaviors are usually of the form that one action step leads to another. Temporal logic is often used to formalize this kind of properties. According to Gabbay [57], specifications in temporal logic can be re-written into the *executable form*, which states "If A holds in the past, then do B", where A and B are some system actions. Unlike [57], in which the states in the system models are treated different according to the current time point, the past are considered declarative and the future are imperative, we consider the whole system model as declarative, and check the properties in the model, because the system implementation is already available when we apply the checking method.

We present here the standard linear temporal logic for expressing the properties of the system behaviors. This linear temporal logic has two fragments, one is the past linear temporal logic (PLTL), and the other is the future linear temporal logic (FLTL), each with different temporal modalities. In the rest of the chapter, we refer the future linear temporal logic as linear temporal logic (LTL) for reasons of simplicity, unless specifically described.

2.3.1 Linear Temporal Logic

The syntax and semantics of linear temporal logic with past and future modalities are given in Definition 2.3.1 and Definition 2.3.2 respectively.

An atomic formula is either p from the propositional variables or the constant *true*. For any formula φ, we can negate it with $\neg$ and get a new formula $\neg\ \varphi$, read as "not φ". Two formulas φ, ψ can be conjuncted with the operator $\wedge$ to $\varphi\ \wedge\ \psi$, read as "φ and ψ".

The above part forms the non-temporal part, and is used in both FLTL and PLTL. There are two temporal operators for PLTL and two corresponding ones for FLTL. For FLTL, $\bigcirc$ is read as "next", and $\mathcal{U}$ is read as "until". This can be used to describe properties like "the ballot casting machine stays locked until a voter inserts the smartcard and enters the correct PIN code" $\bullet$ is read as "previous", and $\mathcal{S}$ is read as "since", which can be used to describe requirement such as "since the incorrect PIN code was entered twice, the smart could not be used anymore".

Definition 2.3.1 (Syntax of Future Linear Temporal Logic).

$$\varphi,\ \psi ::= true \mid p \mid \neg\ \varphi \mid \varphi\ \wedge\ \psi \mid \bigcirc\varphi \mid \varphi\,\mathcal{U}\,\psi$$

Definition 2.3.2 (Syntax of Past Linear Temporal Logic).

$$\varphi,\ \psi ::= true \mid p \mid \neg\ \varphi \mid \varphi\ \wedge\ \psi \mid \bullet\varphi \mid \varphi\,\mathcal{S}\,\psi$$

Next, the standard path semantics [75] for the logic is provided. We denote $\pi = \pi_0, \pi_1, \ldots, \pi_n$ as a path, and L as a function mapping from a path node π_i to a set of atomic propositions $L(\pi_i)$. The model relation is defined as $\pi, i \models \varphi$, where $0 \leq i \leq n$ and φ is a future/past linear temporal formula. We denote the language of φ as $\mathcal{L}(\varphi)$ defined as the set of all π such that $\pi, 0 \models \varphi$.

Definition 2.3.3 (Semantics of Future Linear Temporal Logic).

$\pi, i \models true$	`iff TRUE`
$\pi, i \models p$	`iff` $p \in L(\pi_i)$
$\pi, i \models \neg\ \varphi$	`iff` $\pi, i \not\models \varphi$
$\pi, i \models \varphi \wedge \psi$	`iff` $\pi, i \models \varphi$ `and` $\pi, i \models \psi$
$\pi, i \models \bigcirc\varphi$	`iff` $i < n$ `and` $\pi, i+1 \models \varphi$
$\pi, i \models \varphi\,\mathcal{U}\,\psi$	`iff` $\exists j \geq i. \forall i \leq k < j. \pi, k \models \varphi$ `and` $\pi, j \models \psi$

Definition 2.3.4 (Semantics of Past Linear Temporal Logic).

$\pi, i \models true$	`iff TRUE`
$\pi, i \models p$	`iff` $p \in L(\pi_i)$
$\pi, i \models \neg\ \varphi$	`iff` $\pi, i \not\models \varphi$
$\pi, i \models \varphi \wedge \psi$	`iff` $\pi, i \models \varphi$ `and` $\pi, i \models \psi$
$\pi, i \models \bullet\varphi$	`iff` $\pi, i-1 \models \varphi$
$\pi, i \models \varphi\,\mathcal{S}\,\psi$	`iff` $\exists 0 \leq j \leq i. \forall j < k \leq i. \pi, k \models \varphi$ `and` $\pi, j \models \psi$

Definition 2.3.5 (Syntactic sugar for Linear Temporal Logic).

$$\begin{array}{rlrl} \varphi \vee \psi \equiv & \neg(\neg\varphi \wedge \neg\psi) & \varphi \rightarrow \psi \equiv & (\neg\varphi) \vee \psi \\ false \equiv & \neg true & \Diamond\varphi \equiv & true\,\mathcal{U}\,\varphi \\ \Box\varphi \equiv & \neg(\Diamond\neg\varphi) & \varphi\,\mathcal{R}\,\psi \equiv & \neg(\neg\varphi\,\mathcal{U}\,\neg\psi) \end{array}$$

It's easy to see that properties in mode 1 and mode 2 correspond to PLTL and FLTL respectively. For example, "After the server receives a ballot message from the voter, the sever should validate it" is a property in

Mode 1, and it can be formulated as $receiveBallot \rightarrow \Diamond validateBallot$, "If a sever validates a ballot, then the ballot must came from a message from the voter" is a mode 2 property.

Past-time modalities do not add expressiveness to future-time linear temporal logic [58], but past-time modalities are used to express properties more succinctly, and [91] shows the gap of the transition between LTL and FLTL is at least single exponential. In addition, for our purpose of application, the succinct expression is important for formalizing properties.

Before we move to the translation from LTL to automaton, we show that the model relation of PLTL can be translated into the model relation FLTL. Next we show an important feature between the past and future LTL. A function $\{||\}$ is provided to map PLTL formula to LTL formula, and it is defined in Definition 2.3.6

Definition 2.3.6 (Converse Formula from PLTL to FLTL).

$$\begin{array}{lll}
\{|true|\} & \text{if } P & \text{is } true \\
\{|p|\} & \text{if } P & \text{is } p \\
\{|\neg\ \varphi|\} & \text{if } P & \text{is } \neg\ \{|\varphi|\} \\
\{|\varphi\ \wedge\ \psi|\} & \text{if } P & \text{is } \{|\varphi|\}\ \wedge\ \{|\psi|\} \\
\{|\bullet\varphi|\} & \text{if } P & \text{is } \bigcirc\{|\varphi|\} \\
\{|\varphi\ \mathcal{S}\ \psi|\} & \text{if } P & \text{is } \{|\varphi|\}\ \mathcal{U}\ \{|\psi|\}
\end{array}$$

We use the symbol $\{||\}$ for path π as well, $\{|\pi|\} = \pi_n, \pi_{n-1}, \ldots, \pi_0$, where $\pi = \pi_0, \ldots, \pi_{n-1}, \pi_n$

Theorem 2.3.1 shows that for all PLTL formula checking can be done via LTL checking, thus we only need to focus on the method of checking LTL formulas.

Theorem 2.3.1 (Symmetricity between PLTL and FLTL)
For any PLTL formula φ and its FTLT converse $\varphi' = \{|\varphi|\}$, the checking of a path π for φ yields the same result as checking π's reversed path $\pi' = \{|\varphi|\}$ with φ', which means $\pi, i\ \models\ \varphi$ if and only if $\pi', (n-i)\ \models\ \varphi'$

Proof. By induction on the formula φ. For example, it's easy to show that $\pi, i\ \models\ \bullet\varphi$ if and only if $\pi', (n-i)\ \models\ \bigcirc\{|\varphi|\}$ provided that $\pi, (i-1)\ \models\ \varphi$ if and only if $\pi', (n-i+1)\ \models\ \{|\varphi|\}$. □

2.3.2 Büchi Automata

The linear temporal logic formula can be translated to Büchi Automaton. A Büchi Automaton is defined as $\mathcal{A} = (Q, \Sigma, I, T, F)$, where Q is a finite set of states, Σ is the input (alphabet), $I \subseteq Q$ is the set of initial states, $T \subseteq Q \times \Sigma \times Q$ is the set of transitions, and $F \subseteq Q$ is the set of accepting states.

Linear temporal logic with future-time operators has been studied intensively. As discussed in the related work section, there are many algorithms that have been proposed in study of the relation between temporal logic and Büchi automata. We describe briefly the future-time linear temporal logic algorithm from [61] by Gastin and Oddoux. There have been more efficient solutions like [49], but they are not within the scope of this chapter.

Gastin's translation algorithm consists of three steps.

1. Pre-processing the original formula to negation normal form φ
   The pre-processing consists of two steps, one is rewrite formulas into negation normal form, in which the negation is only directly applied to predicates. Based on the equivalence relation of the syntactic sugar, the negation normal form of the formulas can push the negation operator until the predicate level.

2. Translate φ into a generalized Büchi automaton $\mathcal{A}'_\varphi$ with the help of Very Weak Alternating Automata (VWAA).
   From a pre-processed formula φ, a generalized Büchi automaton (GBA) $\mathcal{A}'_\varphi$ can be constructed. This construction has two steps, LTL to very weak alternating automata (VWAA), and VWAA to generalized Büchi automata.

3. Post-processing $\mathcal{A}'_\varphi$, simplify and translate it into a Büchi automaton $\mathcal{A}'_\varphi$.

The proof of the correctness theorem of the translation, Theorem 2.3.2, can be found in [61]. The theorem states that the accepted input for the automaton are the same as that which models the formula. We use $\mathcal{L}(\mathcal{A})$ to denote the accepted language of the automaton. This is the fundamental correctness theorem for our method discussed in the next section. With this theorem, we can translate the model relation problem into the Büchi automaton acceptance problem. Later we will show how the finite sate in Büchi automaton helps to reduce the checking spaces.

Theorem 2.3.2
The Büchi automaton $\mathcal{A}_\varphi$ accepts precisely the models of φ, i.e.

$$\mathcal{L}(\mathcal{A}_\varphi) = \mathcal{L}(\varphi)$$

Definition 2.4.1 (An Abstraction Language)**.**

$$\begin{aligned}
\text{Method} &::= \text{Identifier}\texttt{(}\text{Expression } (\texttt{,}\ \text{Expression})^{*}\texttt{)} \mid \text{Identifier}\texttt{()}\\
\text{Expression} &::= \text{Expression } (\texttt{\&\&}, \texttt{||}, \texttt{==}, \texttt{>}, \texttt{<}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/})\ \text{Expression}\\
&\quad\mid \text{Method} \mid \text{Identifer} \mid \text{IntegerLiteral} \mid \texttt{true} \mid \texttt{false}\\
\text{Statement} &::= \texttt{if (}\text{Expression}\texttt{)}\ \text{Statement } \texttt{else}\ \text{Statement}\\
&\quad\mid \texttt{while (}\text{Expression}\texttt{)}\ \text{Statement}\\
&\quad\mid \text{MethodMethod}\texttt{;}\\
&\quad\mid \text{Expression}\texttt{=}\text{Expression}\texttt{;}\\
&\quad\mid \texttt{\{}(\text{Statement}) * \texttt{\}}
\end{aligned}$$

2.4 An abstraction Language

2.4.1 Language

Checking programs using automata has to make use of two components, the automaton and the input for the automaton. In this section, we show how to extract the input to the automaton from the system that needs to be verified. We give the grammar of an abstraction language which represents the content of a function, and then show the abstract input to the automaton can be represented by a set of paths. This abstraction language is an abstraction from practical programming languages including only some essential components, which means the representation is suitable for many programming languages. But we will show later that this abstraction language requires more effort to apply then those with more details.

An *identifier* is a sequence of letters, digits, and underscores, starting with a letter. An *integerliteral* is a sequence of decimal digits that denotes the corresponding integer value, including negative sign. Reserved symbols are denoted in red. "*" means repetition, and "()" groups items together.

There are some restrictions in this grammar. In object-oriented languages, this simplified grammar doesn't differentiate method (function) calling in classes with ".", and these calls will only be recognized as a special identifier. So we need to use an external way to handle classes and other types, which will be discussed in Section 2.5.2. Other loops, like `for` loop and `do ...while` loop can be reformed in to the `while` loop without any difficulty.

Next, we define the path generated from that abstraction language. We focus on the function calling (method invocation) part of the language, and we also assume the input to the Büchi automaton is a list of method names (identifiers), which means there's a mapping between the logic predicate and the function names. We assume that both branch of `if` statement are covered, which means there should be no dead code in the if branch. The path can later be extended to path with variables and data sensitivity.

Definition 2.4.2 (Function Path)**.**

$$\begin{aligned} \text{F} ::= & \text{Identifier F} \mid \texttt{Nil} \\ \text{S, S'} ::= & \texttt{ifl}\ F\ S \\ & \mid \texttt{ifr}\ F\ S \\ & \mid \texttt{whl}\ F\ S \\ & \mid F \\ & \mid S\ S' \end{aligned}$$

In order to obtain the paths, we define an auxiliary function $[\![\cdot]\!]$, which maps the program code (either *Statement*, *Expression* or *Method*) to the path S (also denoted as π) defined above. By keeping the only type of terminal symbols (method identifier), we get a set of paths as input for the automaton.

Definition 2.4.3 (Path Generating Function $[\![P]\!] =$).

Method

$\{$Identifier `Nil`$\}$	if P is Identifier`()`
$\{$Identifier $[\![E0]\!]$ ...$\}$	if P is Identifier(E0, ...)

Expression

$[\![MethodInvocation]\!]$	if P is a $MethodInvocation$
$[\![E_1]\!] \times [\![E_2]\!]$	if P is $E_1($`&&`$, ...)E_2$
$\{$`Nil`$\}$	if P is other Expressions

Statement

$\{$`ifl` $[\![E]\!]$ $[\![S_1]\!]$; `ifr` $[\![E]\!]$ $[\![S_2]\!]\}$	if P is `if` (E) S_1 `else` S_2
$\{$`whl` $[\![E]\!]$ $[\![S]\!]\}$	if P is `while` (E) S
$[\![MethodInvocation]\!]$	if P is $MethodInvocation$ `;`
$[\![E_1]\!] \times [\![E_2]\!]$	if P is E_1`=`E_2 `;`
$[\![S_0]\!] \times [\![S_1]\!] \times ...$	if P is $\{S_0; S_1; ...\}$

$[\![S0]\!] \times [\![S1]\!]$ is a path concatenation in the style of Cartesian product, for example $\{p_1; p_2\} \times \{p3\}$ is $\{p_1\ p_3; p_2\ p_3\}$. It is easy to check that the branching of path only happens in the `if` statement.

For the `whl` path node, we have to check all possible numbers of loops, because there's no way to determine the number of runs correctly in all possible `while` statement without further analysis on the semantic level. We prove in Theorem 2.4.1 that the maximum number for looping is bound by the number of states in the automaton. So the {`whl` E S} paths set becomes {E; E S E; E S E S E; E S E S E S E; ...} which repeats S E until the number of states in the automaton $\mathcal{A}_\varphi$.

Theorem 2.4.1
For the unrolling of the $\{$`whl` $E\ S\}$ *statement, the maximum number of depth is the number of states in the automaton* $\mathcal{A}$.

Proof. This proof is based on the pigeon hole principle. First, we group input within the `whl` statement as a single transition together, because when the program runs the `while` statement, all inputs are executed as a group. Here we only consider the case that the automaton will not end up stuck due to the input leading to no transition at all, otherwise the automaton will surely not end up in an accepting state. From any state in $\mathcal{A}$ after the total number of state transitions, there will be a state which has already appeared in its previous transition. Therefore, any further unrolling will have a same result shown with less number of unrolling. $\square$

After the unroll of the `whl` statement, all paths are branching and loop free. The checking function takes in all the paths, and runs each of them as an input to the Büchi automaton. The result of this checking is true if it ends in an accepting state otherwise false, unless the rejected path has another path which is unfolded from the same loop that was accepted. This means we are checking the existence of the correct path.

Theorem 2.4.2
The set of paths $\Pi = [\![P]\!]$ captures the abstraction program P's all method calls (identifier).

Proof. This can be proved by induction on the program P. Because the method calls will only appear in $Method$ according to the grammar, and all $Method$ is captured by $[\![\cdot]\!]$, all method calls (identifiers) are captured in the paths $\Pi = [\![P]\!]$. $\square$

2.4.2 Properties

There are several abstraction levels: from the implementation source code to the abstraction language program, from the abstraction language program to the class path. The correctness and termination of the abstraction from the implementation language to the abstraction language heavily depends on the code scanner tool that is used, and more is discussed in Section 2.5.2. The second part of the abstraction is discussed here, and some theorems for termination and abstraction of this checking method are presented.

Theorem 2.4.3 (Termination of Path Generation)
The checking of an automaton $\mathcal{A}$ will terminate with generated path π as input.

Proof. The idea of the proof is that, trees of the program execution path are at most finite branching with `if` statement, but (possibly) with infinite depth through `while` loop. Theorem 2.4.1 has proved the infinite depth in `while` loop can be unrolled into finite paths without losing any trace of checking the program paths. Thus the total number of paths for the input will be finite, and all these paths are finite, which means the checking is actually the finite input for an automaton, then leads to the termination. □

Definition 2.4.4 (Path Acceptance). A path π is accepted by the automaton if and only if at least one of the path's unrolled (by unrolling `whl` loop) paths is accepted by the automaton.

Theorem 2.4.4 (Abstraction Theorem)
If the specification policy φ is followed in program P up to the abstraction level, i.e. only function calls are considered, then paths generated from program P are accepted by the automaton $\mathcal{A}_\varphi$.

Proof. The idea of the proof is that, all possible execution paths of the program are checked if they follow the policy φ. The only difference between the execution paths and the abstract paths is the `whl` loop, which is checked for existence based on Theorem 2.4.1. If the φ is followed in the source code up to the function call level, then path π generated by $[\![]\!]$ should accepted by $\mathcal{L}(\varphi)$. Then by theorem 2.3.2, we have π is also accepted in $\mathcal{A}_\varphi$. □

Theorem 2.4.4 shows that our abstraction is TI abstraction [62]. We would like to remark that our techniques works for any TI abstraction, which means, that more information the generated paths contain, the more precise the analysis will.

2.5 Application

2.5.1 Source Code Analyzer

The typical way the source code analyzers may be used for static analysis consists of the following steps. The source code is first translated into an intermediate model, which is optimized for later analysis. During this translation, the code scanner tool generates an abstract model from the source code. In the traditional setting of a source code analyzer, like Coverity, some security policies are predefined, and users can use the checker to analyze the software via pattern matching these predefined behaviors in the source code. For example, after a resource is required, there should always be a release happening for this resource after it's required. If a hazardous behavior pattern shows up in the source code, such as the release of a resource never happens, the analyzer will generate some error or warning messaging.

These patterns predefined in the analyzer are also called non-functional properties, which means software systems should generally should follow these guidelines to avoid some common mistakes in software engineering. These patterns and checks includes inputs handling, buffer overflow, error and exception handling, and so on. For the purpose of our evidence flow analysis, we focus on the functional properties. The functional properties are system specific, such as the properties defined in the system specification. With different code scanning tools, multiple programming languages are supported, which means our application is not restricted to one single programming language.

After specifying the properties, the problem now lies in the building of the model. The abstraction level of the model heavily influences the complexity and the result of the checking. We use a very high abstract level, function calls only, for the modeling as shown above. When the abstract syntax tree is built, the execution path of the implementation is also provided as the traversal from root to end nodes. This opens opportunities to design and create our own checker on top of many off the shelf code and functionalities. The paths can be very helpful to check some system properties. In this chapter, we use the Coverity analyzer to generate the needed input for us.

2.5.2 Coverity

We conduct our test with the SDK extension from Coverity, and use the LTL2BA tool to generate the Büchi automaton from LTL properties. The syntax of the automaton is in PROMELA, a modeling language introduced by Gerard J. Holzmann in SPIN [74]. We build the automaton from the PROMELA file, as shown in 2.6.1, in Coverity SDK, and checking the properties by feeding the paths as input.

There are two possible ways to run the model checking, one is the original property, and the other is the negation. Though, the negation one can sometime be faster and terminate earlier than checking the original formula directly. For simplicity reason, we show the original one as example.

The evidence flow as we discussed is closely related to *taint analysis*. Taint analysis usually does the following: it tracks a sensitive data and runs the data flow analysis to see if the tainted data ends up in a hazardous state. Some related actions can be defined for the evidence, including define the source (starting point of the evidence) and the sink (end point of the evidence), add the pass through action (passing the evidence to another function or variable) and the cleanse action (after which, the data is considered non sensitive). These actions can be mapped to the function call handling, for example, the source can be defined as the message receive action, and the sink can be the signature or error send back action. The pass through action can be seen as the variable containing the sensitive data (e.g. a ballot) passing to another variable (e.g. ballot signature). We have not obtained the license to use Coverity's taint analysis, therefore we had to cut corner and to implement our method with the idea from taint analysis.

Coverity's code scanning is based on the abstract syntax tree, and its store based checking and type based checking. The abstraction from the source code to the abstract language, and further the path are also implemented in Coverity SDK. The store based checking is path sensitive, but can not expend between different function calls. However, the type based checking can be used to propagate function calls path by flatten some functions. Each path in the abstract syntax tree is considered as an input path for the automaton for the property checking. The auxiliary function discussed in Section 2.4.1 is implemented in Coverity SDK. In the implementation, we extract the path by pattern matching function calls, also check if they are in loops or not. All loops will be unrolled in the end according to the number of states in the automaton generated from the property.

2.5.3 Practical Considerations

We make use of Coverity for getting the abstract path from the Java source code for vVote, but there are several problems needed to be handled. The Coverity SDK provides a path sensitive way to extract function paths and a set pattern matching functions to retrieve the method names. But when we want to get a complete execution path, instead of a single path in one function, we will need to flatten some of the function calls to its execution path by replacing the function call with the set of paths that function contains.

Flatten out method calls

We flatten the method call up to the level that the function name is also used as a predicate, and set a maximum number for unnecessary flattening. For unique declaration, this should not be any problem, but for the case with method override and abstract method, this is more complicated. The flatten method is based on the type of the function, and the type information is generated from the type based checking in Coverity, which generates the parent class and methods of each method. We used Coverity to generate the necessary type information, and flatten the functions with the help of the type information from the functions.

Multi Threads

Our method does not cover the multi thread case, which means that the interaction between threads will not be captured automatically. So we handle each thread manually with hard coded checking information, such as checking the “run” method in the related thread. More detail is given in the Victoria example in Section 2.6.2.

2.6 Case Study

2.6.1 A Simple Example

Actions have some dependency on each other in a temporal order, and we show here how it can be expressed in LTL and then checked. We present a property, as a simple example, that is close to the Victorian case study in Section 2.6.2.

Example 2.6.1. Formula $\varphi = \Box(checkVote \rightarrow \Diamond sendConfirm)$ means whenever a check vote action happened, a confirmation should be send back. Formula $\psi = \blacksquare(sendConfirm \rightarrow \blacklozenge checkVote)$, means whenever a confirmation is sent, there should be a check vote that happened before. We provide details of checking φ, and the formula ψ can be checked using the converse method discussed in Section 2.3.

First, formula φ will be translated into a Büchi Automaton $\mathcal{A}_\varphi$ as shown in Figure 2.1. There are two states, ”init” state and ”1” state, where both the initial and the accepting state is the ”init” state. The transition labels are the predicate names, and ”1” in the label matches any predicate names.

”!checkVote” means any predicate but `checkVote`, and `a || b` is used used to match either `a` or `b`.

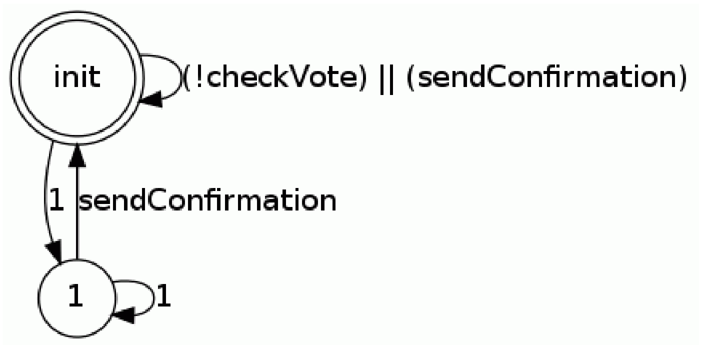


Figure 2.1: Büchi Automaton Example

The LTL2BA generated Büchi automaton code in PROMELA is shown as following.

```
never { /* G ( checkVote -> F sendConfirm ) */
accept_init :    /* init */
        if
        :: (!checkVote) || (sendConfirm) -> goto accept_init
        :: (1) -> goto T0_S2
        fi;
T0_S2 :    /* 1 */
        if
        :: (1) -> goto T0_S2
        :: (sendConfirm) -> goto accept_init
        fi;
}
```

Next, we take a piece of code as an example for the property checking.

Example 2.6.2. A simple piece of code

```
checkVote();
sendConfirm();
checkVote();
while (a_boolean_value) {
  sendConfirm();
}
```

The function names $F =$ `checkVote | sendConfirm | checkVote`

After applying $[\![\cdot]\!]$ to the code above, the returned path is {`checkVote, sendConfirm, checkVote, (whl Nil sendConfirm)`}.

The input for Büchi Automaton $\mathcal{A}_\varphi$ is the set of path {`checkVote, sendConfirm, checkVote; checkVote, sendConfirm, checkVote, sendConfirm; checkVote, sendConfirm, checkVote, sendConfirm, sendConfirm`} by unrolling the `whl` statement up to two times, and the decoration symbol `whl` and the empty symbol `Nil` will be gone after the unrolling.

The second and the third path of the input set end in the "init"' state, which is the accepting state, thus the policy is followed. In contrast, paths from the following code will not be accepted, thus the property is not followed. Because all paths end up in the non-accepting state "1" after the unrolling.

```
checkVote();
while (a_boolean_value) {
  sendConfirm();
  checkVote();
}
```

2.6.2 State of Victoria

The Australian state of Victoria used a variant of the Prêt à Voter [31, 45] cryptographic voting protocol for the 2014 state election. There are many components involved in the system, such as the POD which prints the empty ballot, EBM (also called EVM) which helps voter to vote, public bulletin board which publishes information for receipt checking, and private bulletin board (WBB) which acts as the server and is in charge of coordinating the ballot generating, ballot casting and receipt generating.

We take the private Bulletin board's ballot receiving process as a case study. WBB contains multiple peers. Communication between peers is considered internal, and communication between a WBB peer and EBM is external. All income and outcome packets for each WBB peer are treated differently depends on the packet type, such as the "StartEVMMessage" and the "VoteMessage".

Figure 2.2 shows the diagram for handling the vote message from EBM in a WBB peer. We can use the following formula to formulate this speci-

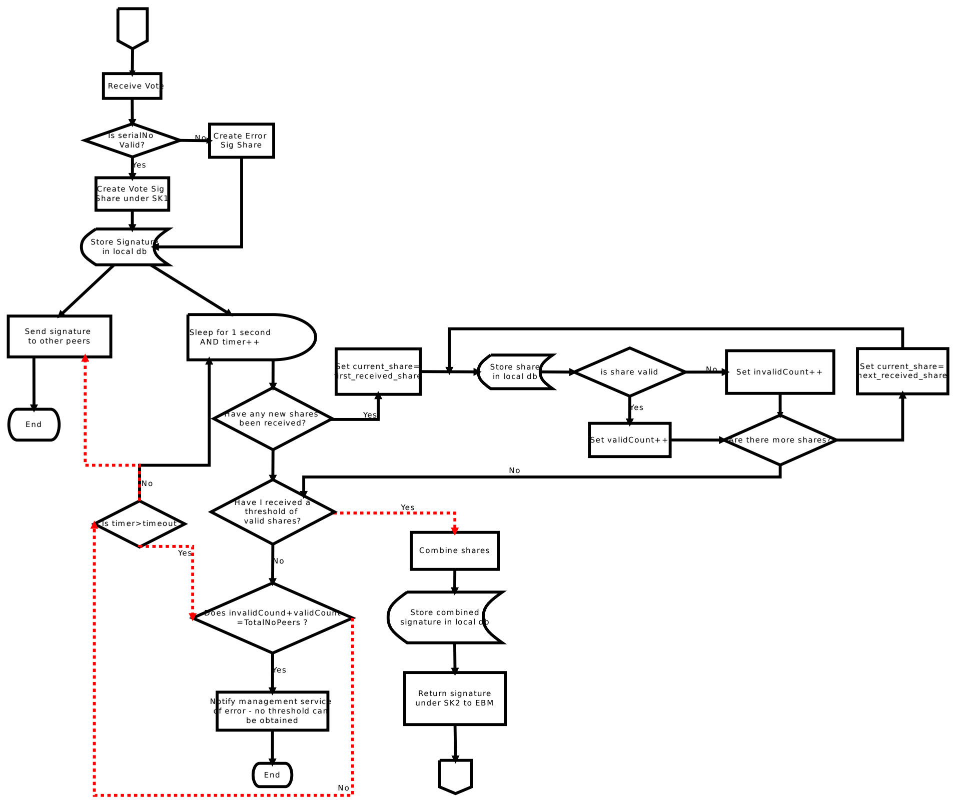

Receive Vote
Is serialNo Valid?
No
Create Error Sig Share
Yes
Create Vote Sig Share under SK1
Store Signature in local db
Send signature to other peers
End
Sleep for 1 second AND timer++
Have any new shares been received?
Yes
Set current_share= first_received_share
Store share in local db
Is share valid
No
Set invalidCount++
Set current_share= next_received_share
Yes
Set validCount++
Are there more shares?
No
No
Is timer>timeout
Yes
Have I received a threshold of valid shares?
Yes
Combine shares
No
Does invalidCound+validCount =TotalNoPeers ?
Store combined signature in local db
Yes
Notify management service of error - no threshold can be obtained
Return signature under SK2 to EBM
End
No

Figure 2.2: EBM Message in WBB [44]

fication in Mode 1. Similarly, the specification in mode 2 can be defined.

1. $\Box(ReceiveVote \rightarrow \Diamond CheckSerialNo)$
2. $\Box(CheckSerialNo \rightarrow \Diamond(CreateSigSK1 \vee CreateErrorSig))$
3. $\Box((CreateSigSK1 \vee CreateErrorSig) \rightarrow \Diamond StoreSiginDB)$
4. $\Box(StoreSiginDB \rightarrow (\Diamond SendPeerSig \wedge \Diamond Sleep))$
5. $\Box(Sleep \rightarrow \Diamond(CheckNewShares))$
6. $\Box(CheckNewShares \rightarrow \Diamond(ReceiveThreshold \vee SetCurrentShareFirst))$
7. $\Box((SetCurrentShareFirst \vee SetCurrentShareNext) \rightarrow \Diamond(StoreShareinDB))$

8. $\Box(StoreShareinDB \ \rightarrow \ \Diamond(IsShareValid))$

9. $\Box(IsShareValid \ \rightarrow \ \Diamond(SetInvalidCount \vee SetValidCount))$

10. $\Box((SetInvalidCount \vee SetValidCount) \ \rightarrow \ \Diamond CheckMoreShare)$

11. $\Box(CheckMoreShare \ \rightarrow \ \Diamond(SetCurrentShareNext \vee ReceiveThreshold))$

12. $\Box(ReceiveThreshold \ \rightarrow \ \Diamond(TotalPeerNoReceive \vee CombineShare))$

13. $\Box(CombineShare \ \rightarrow \ \Diamond StoreCombinedSig)$

14. $\Box(StoreCombinedSig \ \rightarrow \ \Diamond ReturnSigSK2)$

15. $\Box(TotalPeerNoReceive \ \rightarrow \ \Diamond(TimeOut \vee ThresholdFailed))$

16. $\Box(TimeOut \ \rightarrow \ \Diamond(ThresholdFailed \vee (\Diamond Sleep \wedge \Diamond SendPeerSig)))$

2.6.3 Test and Findings

We specified some of the flatten steps in our implementation to generate the path in order to avoid unroll too many non-necessary overridden functions, and then checked the aforementioned properties for EBM message. For example, when “External Message” is received, for our checking purpose, we take the “Vote Message”, a subclass of “External Message” to flatten the method. The type of the message is run-time dependent, it checks the incoming external message from the ballot marking machine, and parses the message in JSON format, in which the type field of the message is included. The whole procedure contains the following three threads: the external message receive thread, the message process thread, and the timeout thread.

Method “WBBExternalPeerThread.run()” is in the external message thread, in which method “TimeoutManager.addDefaultTimeout(Runnable task)” is used in the external peer thread to handle timeout, and the method “SerialExecutor.execute(Runnable r)” is used to process the message in message process thread.

The core process part is the message process thread, the external message thread takes care of read in message, and validates the message. However these threads do not run independently, the external peer thread initiates the timeout thread, and starts the process message before it ends. We concentrate the flattened run() function in these three threads, and then unfold them to check if the design specification is followed.

We managed to construct execution traces based on the unique unmangled name of the method and some hard coded flattening of the functions. It takes about 2.9 seconds to generate paths in WBB, which includes 88 java files and 99 classes, and there are 1142 paths consider different functions individually before flattening.

Here we give some examples of how property checking is conducted.

In the design, it is required that before any processing of the message, the message itself should be validated first. The following property states that every message should be validated before it gets processed.

$$\blacksquare(preProcessMessage \rightarrow \blacklozenge performValidation)$$

We check the formula through its converse, which is

$$\square(preProcessMessage \rightarrow \lozenge performValidation)$$

, with a reversed order of the paths in class `WBBExternalPeerThread`

One challenging for tracking evidence flow by function calls is that not all steps for the evidence go through the function calls. This means not all formulas described in the previous subsection are able to map into function call representation. For example, the "TotalPeerNoReceive" predicate won't appear in the path, because it is represented by a return value from checking the threshold.

Here, we provide a list of mapping examples between the function names in the source code and the predicate in the formulas in Table 2.1.

Predicate	Function Name
ReceiveVote	parseMessage
CheckSerialNo	performValidation
CreateSigSK1	createInternalSignature
StoreSiginDB	storeIncomingMessage
SendPeerSig	sendToAllPeers
ReceiveThreshold	checkThreshold
TimeOut	checkAndSetTimeOut
Sleep	sleep
ThresholdFailed	canOrHaveReachedConsensus
TotalPeerNoReceived	canOrHaveReachedConsensus
CombineShare	constructResponseAndSign

Table 2.1: Function Names and Predicates

We describe here two policies that are violated.

1. $\Box(TimeOut \rightarrow \Diamond(ThresholdFailed \vee (\Diamond Sleep \wedge \Diamond SendPeerSig)))$

2. $\Box(ReceiveThreshold \rightarrow \Diamond(TotalPeerNoReceive \vee CombineShare))$

The first formula states that, when timeout is checked, it either send the signatures again to other peers and goes to sleep for a while, or goes to the threshold failed state. In our test, this policy is not followed, because the timeout thread directly goes to send the error back in the source code instead of checking the threshold once the time is out. By looking back at the design, we also notice that there is a problem in the design. The design forms a possibly infinite loop with checking the threshold count after time is out, then it goes back to timeout again if the total number of peer's signature is not received. Meanwhile in the code, timeout checker terminates the loop when time is out. The related path of these actions are marked as dotted red in Figure 2.2.

The second policy above is also not followed, because the message process thread ends directly after checking the threshold. The implementation handled this in a listener, which processes the peer's signature share message separately. This violation is caused by the difference between the actual implementation and the design specification.

We remark that, because the source code uses `try...catch` block to capture exception all the time, there are plenty of exit points before they continue with the route shown in Figure 2.2, which also leads to other difference between the design and the implementation.

Chapter 3

Epistemic Policies for Voting Systems[1]

3.1 Introduction

Chapter 2 provides a method to verify property in a static way on source code. This chapter, we turn our attention to property verification based on log checking.

In modern voting systems, disputes regarding claims that something went wrong during an election are usually settled by inspecting logs, as these are often the only persistent records of the events that occurred during operation. Typical challenges include that a vote was not properly counted, that it was changed by and adversary, that the election commission permitted voters to vote multiple times, or that the secrecy of a vote was compromised. Voter-verifiable systems provide additional mechanisms to allow voters to check if his or her vote was cast-as-intended and/or counted-as-cast also rely on logs when it comes to settling claims, for example, that a server was compromised or when a software "glitch" caused a system to fail.

One way to cope with this challenge is to formulate in a precise manner what a "valid" log should look like by describing its properties in a mathematical concise way. These descriptions are generally referred to as security policies. In this chapter, we study the role of security policies for electronic voting systems, and describe a tool that we developed to analyze them automatically.

[1]Joint work with Daniel Gustafsson and Carsten Schürmann

Most security policy languages are based on logic. Early examples include the Chinese wall security policy [27], security policies in deontic logic [21], and the logic-based security language Binder [50], but there are many others. Security policy languages are usually tailored to their application domains. There are languages that capture time and time-outs [13], access control [60], and aggregation [12].

This chapter presents a model-checker for the epistemic first-order temporal logic (EFOTL) to express security protocols for distributed and communicating systems. EFOTL is an extension of first-order linear time temporal logic with an epistemic connective representing knowledge for specific agents. The hallmark characteristic of EFOTL is a set of modal connectives, also called knowledge modalities, that allow us to express*epistemic security policies* in terms of what each agent knows. Every node in a distributed system, for example servers, client computers, printers, can be considered agents and by using the different knowledge modalities, we can express security policies that involve several agents, concisely and succinctly.

When we try to express global security policies in First-Order temporal logic, we need to do this via user-defined predicates, such as $knows(a, P)$ that capture that an agent a (for example, representing a voter, a database, a printer) knows P. Although this is possible, it tends to lead to complex and convoluted security policies that are difficult to maintain.

The combination of epistemic and temporal features of EFOTL allows us to formulate a large class of important policies for voting systems. Examples of such epistemic security policies include the authentication between two or more principals, commitments to shared-secrets between agents, checks if threshold were reached before decision were made, and protocol conforming communication patterns. Every send instruction initiated by one agent must be matched by a receive instruction of another agent. Ballot papers must be created, stored, retrieved, completed, and submitted or audited, and all events must happen in a particular order.

As case study, we derive epistemic security policies for the Victoria 2014 state election, Australia and analyze the log files that were given to us.[2] Only few voters were permitted to use the machines (voters with disabilities, or voters living in the UK) in this election, and only 1121 ballots were cast during the election. For other reasons, the logs that we were given were from a handful of polling stations and contained data for even fewer (24) votes but were nevertheless extensive in size (5.4 GB). Despite

[2]Access to the log data courtesy of Craig Burton, Victorian Election Commission.

these small numbers, the technology scales reasonably well. We have scaled our experiments to much larger, automatically generated logs (for several thousands of votes). The findings, when this model-checker is applied to the logs that were generate by the vVote system during the Victoria 2014 state election and the possible logs for Norwegian 2013 parliamentary election, is also reported.

The knowledge modalities of EFOTL allow us to express security policies for an entire system, while referring to local knowledge (logs) of each agent. Each agent may run multiple sessions of the same protocol, for example, for the purpose of authentication, voting, cleansing, and mixing. In a slight deviation from the standard semantics, we identify worlds with sessions. The information about which world a log entry is associated with is usually contained within a log, as our case study shows, and can be easily derived. For the purpose of this chapter we therefore assume that every log entry can be uniquely identified as belonging to a particular session.

Especially for voting systems, where the same protocol is executed over and over again, each log may contain repeated sequences of the similar events. In order to exploit the symmetries between different runs of the same protocol, we will show in this chapter that it is sound to reason about equivalent sessions, i.e. sessions that cannot be further distinguished by inspecting the log. For example, consider two agents Alice and Bob and two instances of the same protocol. Alice and Bob exchanges a sequence of messages. If Bob, by inspecting his own log (for the two sessions) cannot distinguish between the messages sent to Alice, he may not identify the two sessions explicitly. Alice on the other hand may have logged more information and thus can identify the two sessions. Therefore in EFOTL reachability relations are indexed by agents, which supports reasoning about sessions modulo reachability.

This chapter describes a language for expressing security policies about different agents and their local logs and a a reachable world semantics, that allows us to identity similar sessions, which is described in Section 3.2, a model-checker for checking epistemic security policies described in Section 3.3, and a technique for turning logs into models, which is described in Section 3.4. Furthermore, we describe a case study using the logs from the Victory 2014 State Election in Section 3.5. In Section 3.6, we show a possible way to include the metric part, which allows us to express security policies using real time constraints, for example if a certain event happened before, during, or after a certain time interval. Finally, we assess conclusions and describe future work in Section 3.7.

3.2 A Language for System Properties

EFOTL extends first-order temporal logic by epistemic connectives indexed by agents. An agent could refer to a particular participant in a communication, a principal, the name of a computer in a distributed system, or possibly even just a process.

We denote *agents* by $a, b, \dots$ and refer to the set of all agents as $\mathbb{A} = \{a, b, \dots\}$. For example, each node of the web bulletin board [46] that was used during the Victoria State 2014 election is for this work is considered an agent. We denote the *time points* by $i, j, \dots$ and refer to the set of all time points as $\mathbb{I} = \{i, j, \dots\}$. Each entry in a log (that is usually time-stamped when the entry was written) corresponds to a time point relative to the other log entries and their time points. Several log entries may be generated by one single protocol run (session). Examples of such sessions include requests to a bulletin board to store a receipt. In this work, we identify sessions with modal worlds. *Worlds* are denoted by $\mathbb{W} = \{w, v, \dots\}$. In Section 3.4 we return to the interpretation of worlds as sessions and explore this connection in detail.

The term algebra of EFOTL is finite and generated from *variables* $\mathbb{V} := \{x_0, x_1, ...\}$ and uninterpreted *constants* $\mathbb{C}$. We may therefore refer to the set of terms as $\mathbb{T} := \{t_0, t_1, ...t_n\}$. We write $\mathbb{P} := \{P_0, P_1, ...\}$ for the set of *predicates*. These predicates are given a priori, and can, for example, be extracted from the log to be checked. To refer to the arity of a predicate P_n, we write $|P_n|$.

Definition 3.2.1 (Syntax of EFOTL).

$$\begin{gathered}\varphi,\ \psi ::= \mathit{true} \mid P_n(t_1, ..., t_{|P_n|}) \mid \neg\ \varphi \mid \varphi\ \wedge\ \psi \mid \exists x.\ \varphi \mid \\ \bullet\varphi \mid \bigcirc\varphi \mid \varphi\ \mathcal{S}\ \psi \mid \varphi\ \mathcal{U}\ \psi \mid \mathcal{K}_a\varphi\end{gathered}$$

In this work, we assume that time is linear and not branching. We identify each entry in the log with a world, a time point and a agent. The set of well-formed formulas that we will use to express security policies below, contain the finite set of predicates $P_n(t_1, ..., t_{|P_n|})$ (including propositions which are 0-ary predicates) and are closed under negation, conjunction, and existential operators as well as linear time operators such as $\bullet\varphi$ and $\bigcirc\varphi$, which states that φ holds at the previous or next time point, respectively. Furthermore the set is closed under $\varphi\ \mathcal{S}\ \psi$ and $\varphi\ \mathcal{U}\ \psi$, which express that φ holds since ψ and φ holds until ψ holds, respectively. Finally, we close the set of well formed-formulas also under the epistemic operator $\mathcal{K}_a\varphi$ (to be read as a knows φ), which allows us to specify security policies that include multiple agents, and multiple local logs.

We say that a formula φ is *closed* if all terms in the predicates from φ are either constants or bound by the existential operator, following standard convention. Below, we will make extensive use of the following connectives defined as syntactic sugar.

Definition 3.2.2 (Syntactic sugar in EFOTL).

$$\begin{array}{rclrcl}
\varphi \vee \psi \equiv & \neg(\neg\varphi \wedge \neg\psi) & \quad & \varphi \rightarrow \psi \equiv & (\neg\varphi) \vee \psi \\
\forall x.\varphi \equiv & \neg\exists x.\neg\varphi & & \Diamond\varphi & \equiv \mathit{true}\, \mathcal{U}\, \varphi \\
\blacklozenge\varphi \equiv & \mathit{true}\, \mathcal{S}\, \varphi & & \Box\varphi & \equiv \neg(\Diamond\neg\varphi) \\
\blacksquare\varphi \equiv & \neg(\blacklozenge\neg\varphi) & & &
\end{array}$$

The connectives $\vee$, $\rightarrow$ and $\Box$ (always in the future), $\blacksquare$ (always in the past), $\Diamond$ (eventually in the future), $\blacklozenge$ (eventually in the past) and the quantifier $\forall$ are defined in a standard classical way.

We begin now with the description of the semantics of EFOTL using a Kripke structure with a possible world model. A Kripke structure is defined as $\mathcal{M} = (\mathbb{W}, \mathbb{D}, \mu, (R_a)_{a\in\mathbb{A}})$, where $\mathbb{D}$ is the *domain*, each $R_a \subset \mathbb{W} \times \mathbb{W}$ describe the *reachability relation* on worlds that belongs to agent a. Recall that different agents may have different views on the worlds.

We write μ for the standard *interpretation of predicates* that associates with each possible world w, time point i, and agent a, a set of instances: $\mu(w, i, a)$. For convenience, we use $\mathcal{M}^i_{(w,a)}(P)$ for $\mu(w, i, a)(P)$. Analogously, we capture variable binding by an valuation function $\nu : \mathbb{V} \rightarrow \mathbb{D}$, which we extends straightforward to terms as $\nu : \mathbb{T} \rightarrow \mathbb{D}$. Terms t_i are mapped to a corresponding element in the domain.

Next, we define *temporal structures* indexed by time point i for world w as $(\mathcal{M},\ \nu,\ w,\ i)$. Based on this, we can now define the relation $\models$, where $(\mathcal{M}, \nu, w, i) \models_a \varphi$ reads as "From agent a's view, φ is true or satisfied, in world w at time point i of structure $\mathcal{M}$". We use $\mathcal{M} \models_a \varphi$ to represent $(\mathcal{M}, \nu, w, i) \models_a \varphi$ for all possible ν, w and i, and $(\mathcal{M}, w) \models_a \varphi$ to represent $(\mathcal{M}, \nu, w, i) \models_a \varphi$ for all possible ν and i.

Definition 3.2.3 (Semantics of EFOTL). The meaning of formula φ is defined as $(\mathcal{M}, \nu, w, i) \models_a \varphi$ in Figure 3.1.

The semantics of the first-order logic fragment for connectives $\neg$, $\wedge$ and $\exists$ is standard. The cases for the temporal connectives are also standard. Note, how $\bigcirc$ (next) and $\bullet$ (past) affect the time point i by moving forwards and

$(\mathcal{M}, \nu, w, i) \models_a \mathit{true}$	iff	$\texttt{TRUE}$
$(\mathcal{M}, \nu, w, i) \models_a P(t_1, ..., t_{\lvert P\rvert})$	iff	$(\nu(t_1), ..., \nu(t_{\lvert P\rvert})) \in \mathcal{M}^i_{(w,a)}(P)$
$(\mathcal{M}, \nu, w, i) \models_a \neg\, \varphi$	iff	$(\mathcal{M}, \nu, w, i) \not\models_a \varphi$
$(\mathcal{M}, \nu, w, i) \models_a \varphi \wedge \psi$	iff	$(\mathcal{M}, \nu, w, i) \models_a \varphi$ and $(\mathcal{M}, \nu, w, i) \models_a \psi$
$(\mathcal{M}, \nu, w, i) \models_a \exists x.\ \varphi$	iff	for some $d \in \mathbb{D}$, $(\mathcal{M}, \nu[x := d], w, i) \models_a \varphi$
$(\mathcal{M}, \nu, w, i) \models_a \bullet\varphi$	iff	$(\mathcal{M}, \nu, w, i-1) \models_a \varphi$
$(\mathcal{M}, \nu, w, i) \models_a \bigcirc\varphi$	iff	$(\mathcal{M}, \nu, w, i+1) \models_a \varphi$
$(\mathcal{M}, \nu, w, i) \models_a \varphi\, \mathcal{S}\, \psi$	iff	for some $j \leq i$ $(\mathcal{M}, \nu, w, j) \models_a \psi$, and for all $k \in [j+1, i+1)$ $(\mathcal{M}, \nu, w, k) \models_a \varphi$
$(\mathcal{M}, \nu, w, i) \models_a \varphi\, \mathcal{U}\, \psi$	iff	for some $j \geq i$ $(\mathcal{M}, \nu, w, j) \models_a \psi$, and for all $k \in [i, j)$ $(\mathcal{M}, \nu, w, k) \models_a \varphi$
$(\mathcal{M}, \nu, w, i) \models_a \mathcal{K}_b(\varphi)$	iff	for all $w' \in \mathbb{W}$, s.t. $(w,\ w') \in R_a$, $(\mathcal{M}, \nu, w', i) \models_b \varphi$

Figure 3.1: Semantics of EFOTL

backwards in time, respectively. As to the semantics for the until-connective $\mathcal{U}$ and since-connective $\mathcal{S}$, it can literally be read as that φ needs to be valid until ψ is valid at j, or that φ is valid since Ψ was valid at j, respectively.

Finally, the semantics of the epistemic connective $\mathcal{K}_b(\varphi)$ is defined based on the possible word semantics. In order to define the meaning of this formula relative to agent a and world w, it is checked in terms of the meaning of φ relative to b and every world reachable from w according to a. This semantics is designed due to the strong connection between the worlds and the agents in logs, which are defined with respect to the truth local to an agent instead of the standard global one. This design allows us to pinpoint problems when a policy fails to check, and to optimize the checking process as shown in Theorem 3.2.2 below.

Depending on the properties of the family of reachability relation $(R_a)_{a \in \mathbb{A}}$ we can classify EFOTL as model logic $S4$ if the each relation is reflexive and transitive, and as $S5$, if each relation is an equivalence relation, i.e., it is also symmetric. We summarize this observation as a theorem:

Theorem 3.2.1
For all formulas φ and ψ, all structures $\mathcal{M}$ and all agents a, b, where each accessible world relation is an equivalence relation.

1. *The distribution property.* $\mathcal{M} \models_b (\mathcal{K}_a(\varphi) \wedge \mathcal{K}_a(\varphi \to \psi)) \to \mathcal{K}_a(\varphi)$
2. *The knowledge generalization rule. If* $\mathcal{M} \models_a \varphi$ *then* $\mathcal{M} \models_b \mathcal{K}_a(\varphi)$
3. *The knowledge of truth property.* $\mathcal{M} \models_a \mathcal{K}_a(\varphi) \to \varphi$
4. *The positive introspection property.* $\mathcal{M} \models_b \mathcal{K}_a(\varphi) \to \mathcal{K}_b(\mathcal{K}_a(\varphi))$
5. *The negative introspection property.* $\mathcal{M} \models_b \neg\mathcal{K}_a(\varphi) \to \mathcal{K}_b(\neg\mathcal{K}_a(\varphi))$

Proof. (of Theorem 3.2.1)

1. If $(\mathcal{M}, w) \models_b \mathcal{K}_a(\varphi) \wedge \mathcal{K}_a(\varphi \to \psi)$, then for all worlds w' such that $(w, w') \in R_b$, we have both $(\mathcal{M}, w') \models_a \varphi$ and $(\mathcal{M}, w') \models_a \varphi \to \psi$, from which we get $(\mathcal{M}, w') \models_a \varphi$. Thus for all w' such that $(w, w') \in R_b$, $(\mathcal{M}, w') \models_a \psi$, therefore $(\mathcal{M}, w) \models_b \mathcal{K}_a(\psi)$.

2. If $(\mathcal{M}, w) \models_a \varphi$ for all worlds w in $\mathbb{W}$, then for all world w and all w' such that $(w, w') \in R_b$, we have $(\mathcal{M}, w') \models_a \varphi$, thus $(\mathcal{M}, w) \models_b \mathcal{K}_a(\varphi)$ for all worlds w.

3. Suppose that $(\mathcal{M}, w) \models_a \mathcal{K}_a(\varphi)$. Because $(w, w) \in R_a$ for R_a is reflexive, we have $(\mathcal{M}, w) \models_a \varphi$.

4. Suppose that $(\mathcal{M}, w) \models_b \mathcal{K}_a(\varphi)$. Consider any w' such that $(w, w') \in R_b$, for all w'' with $(w', w'') \in R_b$, we have $(w, w'') \in R_b$ since R_b is transitive, thus $(\mathcal{M}, w'') \models_a \varphi$, therefore we have $(\mathcal{M}, w') \models_b \mathcal{K}_a(\varphi)$. By the definition of $\models$, we have $(\mathcal{M}, w) \models_b \mathcal{K}_b(\mathcal{K}_a(\varphi))$

5. Suppose that $(\mathcal{M}, w) \models_b \neg\mathcal{K}_a(\varphi)$, then exits w', such that $(w, w') \in R_b$ and $(\mathcal{M}, w') \models_a \neg\varphi$. For all w'' such that $(w, w'') \in R_b$, we have $(w'', w') \in R_b$ since R_b is transitive and symmetric, which means for all w'' such that $(w, w'') \in R_b$, $(\mathcal{M}, w'') \models_b \neg\mathcal{K}_a(\varphi)$. Therefore $(\mathcal{M}, w) \models_b \mathcal{K}_b(\neg\mathcal{K}_a(\varphi))$.

□

From now on, we only consider reachability relations that are indeed equivalence relations. Thus, for our practical purposes, EFOTL is a modal logic S5.

We construct the equivalence relations $(w, w') \in R_a$ if for all time point i, predicate P, $\mu(w, i, a)(P) = \mu(w', i, a)(P)$. By only considering this equivalence relations, we save time by avoiding redudant checking, since the following theorem permits us to exploit these symmetries during model-checking.

Theorem 3.2.2 (Symmetry)
For all formula φ, if the accessible world relation is an equivalence relation created as above, then for all agent a and for all $(w, w') \in R_a$, then $(\mathcal{M}, \nu, w, i) \models_a \varphi$ iff $(\mathcal{M}, \nu, w', i) \models_a \varphi$.

Proof. (of Theorem 3.2.2) Since R_a is symmetric we need to only prove the if direction which we do by induction on the formula. We demonstrate below only the cases which involves worlds here:
If $\varphi = P(t_1, ..., t_{|P|})$, then the implication holds by definition of R_a.
If $\varphi = \mathcal{K}_b(\psi)$, then for all world w_1 such that $(w, w_1) \in R_a$ then $(\mathcal{M}, \nu, w_1, i) \models_b \varphi$. We need to show that for all worlds w_2 such that $(w', w_2) \in R_a$ then $(\mathcal{M}, \nu, w_2, i) \models_b \varphi$. Since R_a is transitive we therefore have $(w, w_2) \in R_a$ and we are done. $\square$

Agents and terms define distinct syntactic categories. Sometimes, it is useful to be able to refer to agents in predicates, for example to express that a message was send to or received from a particular agent. In order to do so, we denote with $\ulcorner a \urcorner$ the constant in $\mathbb{C}$ that corresponds to agent a. $\ulcorner \cdot \urcorner$ is injective.

Example 3.2.1 (Channel Reliability)**.** As our first example, consider a binary communicating systems, with agents a and b. The first property that we wish to discuss is *channel reliability*. Every message sent by a will eventually be received by b.

$$\Box\ \forall m.\ \mathcal{K}_a(\mathit{send}(\ulcorner b \urcorner, m)) \rightarrow \Diamond \mathcal{K}_b(\mathit{receive}(\ulcorner a \urcorner, m))$$

We comment briefly on the choice of predicates. The formula $\mathcal{K}_a(\mathit{send}(\ulcorner b \urcorner, m))$ is true if and only if $\mathit{send}(\ulcorner b \urcorner, m)$ appears in a's log. This means that in the case of *send*, the first argument refers to the receiver, and in the case of *receive* to the sender. In both cases, the second argument refers to the message itself. Without epistemic connectives, *send* and *receive* would have to be designed as tertiary connectives, by respectively adding sender and receiver explicitly. With epistemic connectives, as it is the case in this example, they may remain implicit.

Example 3.2.2 (Channel Authentication)**.** This example is very similar to the previous one. It is the inverse to channel reliability: Any message received by a, must previously have been send by b.

$$\Box\ \forall m.\ \mathcal{K}_a(receive(\ulcorner b \urcorner, m)) \rightarrow \blacklozenge\mathcal{K}_b(send(\ulcorner a \urcorner, m))$$

Extended examples and case studies may require that safety policies quantify of over agents. Our logic currently does not support this, but we believe that it could be easily extended. We leave an extended design to future work.

Example 3.2.3 (Sequentiality)**.** The final example expresses a security policy, which states that no session may overlap. Any session that was started must run to completion before another can be started. The following EFOTL captures this:

$$\Box \forall id. \forall id'. start(id) \rightarrow \neg start(id')\ \mathcal{U}\ end(id)$$

Note, that this policy, for example, prevents deadlocks, because only one session may run at a time.

It may be possible to strengthen EFOTL further, for example, by adding support for uninterpreted function symbols, higher-order quantification, and quantification over worlds. In the interest of brevity and elegance, however, we describe in this chapter only the basic version of EFOTL, which is sufficiently powerful to capture interesting security policies that we wish to express about distributed and communicating systems.

3.3 Finite Model Checking

In the setting of voting systems, logs are always finite, and hence we will restrict our attention in this section to *finite models* only. A consequence is that model checking is decidable, which means that we can always decide if a formula is satisfied in a given model or not.

Definition 3.3.1 (Finite Models)**.** A model $\mathcal{M} = (\mathbb{W},\ \mathbb{D},\ \mu,\ (R_a)_{a\in\mathbb{A}})$ is finite iff $\mathbb{W}$ and $\mathbb{D}$ are finite sets and there exists a time point i such that for all $j > i$ or $j < 0$, agent a, world w and predicate symbol P then $\mathcal{M}^{j}_{(w,a)}(P) = \emptyset$.

Next, we begin the discussion of some basic insights that will help us define the model checking Algorithm 3.2. Let us return briefly to the semantics of EFOTL. To check if a model satisfies a security policy means to unroll the equivalences described in Figure 3.1, until done. This is all straightforward, except for the two temporal connectives such as $\mathcal{U}$ and $\mathcal{S}$ where we will have to guess the correct j. In order to aid with this choice, we compute an upper and a lower *constant point*, or a *window* if you wish, outside of which the security policy expressed as formula φ will have a constant truth-value. The lower constant point is denoted by $\lfloor \varphi \rfloor_{\mathcal{M}}$ and the upper constant point by $\lceil \varphi \rceil_{\mathcal{M}}$.

Definition 3.3.2 (Lower (Upper) constant point)**.** Let φ be a formula and $\mathcal{M}$ be a model then n is a lower (an upper) constant point for φ in $\mathcal{M}$ if forall $i \leq n$ (forall $i \geq n$) then $(\mathcal{M}, \nu, w, n) \models_a \varphi$ if and only if $(\mathcal{M}, \nu, w, i) \models_a \varphi$.

Definition 3.3.3 (Bounds)**.** Let $\mathcal{M}$ be a finite model, φ a formula. We construct a lower and an upper constant points as defined below, where $\lfloor P \rfloor_{\mathcal{M}}$ refers to the earliest time point when P was mentioned in the μ-component of $\mathcal{M}$, and $\lceil P \rceil_{\mathcal{M}}$ is the last.

Formula φ	$\lfloor \varphi \rfloor_{\mathcal{M}}$	$\lceil \varphi \rceil_{\mathcal{M}}$
True	0	0
$P(t_1, ..., t_n)$	$\lfloor P \rfloor_{\mathcal{M}} - 1$	$\lceil P \rceil_{\mathcal{M}} + 1$
$\neg\varphi$	$\lfloor \varphi \rfloor_{\mathcal{M}}$	$\lceil \varphi \rceil_{\mathcal{M}}$
$\varphi \wedge \psi$	$\lfloor \varphi \rfloor_{\mathcal{M}} \sqcup \lfloor \psi \rfloor_{\mathcal{M}}$	$\lceil \varphi \rceil_{\mathcal{M}} \sqcap \lceil \psi \rceil_{\mathcal{M}}$
$\exists x.\varphi$	$\lfloor \varphi \rfloor_{\mathcal{M}}$	$\lceil \varphi \rceil_{\mathcal{M}}$
$\bullet\varphi$	$\lfloor \varphi \rfloor_{\mathcal{M}} + 1$	$\lceil \varphi \rceil_{\mathcal{M}} + 1$
$\circ\varphi$	$\lfloor \varphi \rfloor_{\mathcal{M}} - 1$	$\lceil \varphi \rceil_{\mathcal{M}} - 1$
$\psi\ \mathcal{S}\ \varphi$	$\lfloor \varphi \rfloor_{\mathcal{M}}$	$\lceil \varphi \rceil_{\mathcal{M}} \sqcap \lceil \psi \rceil_{\mathcal{M}}$
$\psi\ \mathcal{U}\ \varphi$	$\lfloor \varphi \rfloor_{\mathcal{M}} \sqcup \lfloor \psi \rfloor_{\mathcal{M}}$	$\lceil \varphi \rceil_{\mathcal{M}}$
$\mathcal{K}_a(\varphi)$	$\lfloor \varphi \rfloor_{\mathcal{M}}$	$\lceil \varphi \rceil_{\mathcal{M}}$

We will use these constant points as upper/lower bounds, when checking a model. They are not tight, but they do the job of limiting the search for j. If we were to take also agents and worlds into consideration, we could achieve much better bounds. This would improve the performance of our model checker, but would lead to a much more unwieldy Definition 3.3.3.

We now prove that these functions indeed will compute a lower and an upper constant point. In order to to be more concise we only write the time point for the Kripke semantic.

Theorem 3.3.1
Let $\mathcal{M}$ be a finite model, and φ be a formula then $\lfloor \varphi \rfloor_{\mathcal{M}}$ is a lower constant point for φ in $\mathcal{M}$.

Proof. (of Theorem 3.3.1) By induction on φ, we only demonstrate the case for $\varphi \,\mathcal{U}\, \psi$, the remaining cases follow in a similar fashion. Let $n = \lfloor \varphi \rfloor_{\mathcal{M}} \sqcup \lfloor \psi \rfloor_{\mathcal{M}}$ and assume $k \leq n$ we need to show that $k \models \varphi \,\mathcal{U}\, \psi$ iff $n \models \varphi \,\mathcal{U}\, \psi$. In the if direction, by the semantics of $k \models \varphi \,\mathcal{U}\, \psi$, there exists an $j \geq k$ such that $j \models \psi$ and the interval $[k, j) \models \varphi$. For this j, either $j \geq n$ in which case $n \models \varphi \,\mathcal{U}\, \psi$ as we want, or $j < n$ in which case by induction hypothesis on ψ we have $\lfloor \psi \rfloor_{\mathcal{M}} \models \psi$ and because $n \leq \lfloor \psi \rfloor_{\mathcal{M}}$ we get $n \models \psi$ and we are done. The only if direction follows in a similar fashion. □

Theorem 3.3.2
Let $\mathcal{M}$ be a finite model, and φ be a formula then $\lceil \varphi \rceil_{\mathcal{M}}$ is an upper constant point for φ in $\mathcal{M}$.

Proof. By induction on φ, this proof is symmetric to the proof of Theorem 3.3.1 . □

We now present our algorithm in Figure 3.2 for checking finite models, and prove some properties of this algorithm. This algorithm follows closely the definition of the semantics from Figure 3.1, and is given as a lazy functional program. Observe, how we use $\lceil \psi \rceil_{\mathcal{M}}$ and $\lfloor \psi \rfloor_{\mathcal{M}}$ in the cases for checking $\varphi \,\mathcal{U}\, \psi$ and $\varphi \,\mathcal{S}\, \psi$ to guarantee termination of the algorithm. The algorithm uses auxiliary functions like `isElem` and `mu`. x ‘`isElem`‘ xs is used for checking if x is an member of the set xs, and `mu` $m\ i\ w\ a\ p$ is used to compute $\mathcal{M}^{i}_{(w,a)}(P)$ defined in the semantics.

Next, we show the partial correctness of the checking algorithm in Figure 3.2 in Theorem 3.3.3.

Theorem 3.3.3 (Partial correctness of Model Checking) *1. If* `check` $\varphi\ \mathcal{M}\ \nu\ w\ i\ a = True$ *then* $(\mathcal{M}, \nu, w, i) \models_a \varphi$.

2. If `check` $\varphi\ \mathcal{M}\ \nu\ w\ i\ a = False$ *then* $(\mathcal{M}, \nu, w, i) \not\models_a \varphi$.

```
check :: Formula → Model → Nu → World → Time → Agent →
Bool
check f m nu w i a = case f of
  Truth       → True
  -- nu: valuation function, mu: get interpretation of predicates
  Prop p ts   → apply nu ts ‘isElem‘ mu m i w a p
  Not p       → not (check p m nu w i a)
  And p q     → check p m nu w i a && check q m nu w i a
  Exists x p → or [ check p m (extend nu x d) w i a
                  | d ← domain m]
  Previous p → check p m nu w (i - 1) a
  Next p      → check p m nu w (i + 1) a
  -- lcp: get lower constant point
  Since p q   → check q m nu w i a
                 || (i > (lcp m q) && check p m nu w i a
                      && check (Since p q) m nu w (i - 1) a)
  -- ucp: get upper constant point
  Until p q   → check q m nu w i a
                 || (i < (ucp m q) && check p m nu w i a
                      && check (Until p q) m nu w (i + 1) a)
  -- rel m a w: get the relation for agent a from world w
  Knows k p   → and [ check p m nu w’ i k | w’ ←
rel m a w]
```

Figure 3.2: Model Checking Algorithm

Proof. (of Theorem 3.3.3) `check` differs from the Kripke semantics in Fig. 3.1 only in the case for $\mathcal{U}$ and $\mathcal{S}$. In the case of $\varphi_1 \,\mathcal{U}\, \varphi_2$ we check every possible time point until $\lceil \varphi_2 \rceil_{\mathcal{M}}$. If φ_2 don't become true until then it will never be true and we can conclude that $\varphi_1 \,\mathcal{U}\, \varphi_2$ will not hold. A symmetric argument is made for $\mathcal{S}$. □

Total correctness of the algorithm in Figure 3.2 is provided by proving that it will terminate.

Theorem 3.3.4 (Termination of Model Checking)
Let φ be a formula and $\mathcal{M}$ be a finite model then `check` $\varphi\ \mathcal{M}\ \nu\ w\ i\ a$ *will terminate.*

Proof. (of Theorem 3.3.4) In the case of $\exists x.\varphi_1$ we are recursing on a subformula, and we know that the domain $\mathbb{D}$ is finite. The case for $\mathcal{K}_a(\varphi')$

is similar, and here we know that $\mathbb{W}$ is finite. The remaining cases either returns immediately or recurse on the subformula, with the exception of $\mathcal{S}$ and $\mathcal{U}$. In the case of $\varphi_1 \,\mathcal{U}\, \varphi_2$ the distance between the current time point i and $\lceil \varphi_2 \rceil_{\mathcal{M}}$ will decrease until $i > \lceil \varphi_2 \rceil_{\mathcal{M}}$. If this point is reached the algorithm immediately returns. The case for $\mathcal{S}$ follows a similar argument. $\square$

This establishes that model checking in finite models is decidable, and we will now address the complexity of algorithm in Figure 3.2. The complexity of the algorithm is shown in Theorem 3.3.5.

Theorem 3.3.5 (Complexity for Model Checking)
Let φ be a formula and M be a finite Kripke model. Given the world w, time point i, and agent a, the decision problem for $(\mathcal{M}, \nu, w, i) \models_a \varphi$ is in $\mathcal{O}(|\mathcal{M}|^{|\varphi|})$. The size of the model $|\mathcal{M}|$ is $|\mathcal{W}| + |\mathcal{D}| + j$, where j is the time point where no predicate are no longer true.

Proof. (of Theorem 3.3.5))
We show this by showing that the Algorithm 3.2 have this complexity, which is done by induction on the structure of φ.

- $P(c_1, ..., c_{|P|})$:, this is a set contain problem, and it requires polynomial time respecting to size of structure.
- $\neg\varphi_1$:, this is a constant time with respect to checking φ_1.
- $\varphi_1 \wedge \varphi_2$:, this is constant to the complexity of checking φ_1 and φ_2.
- $\exists x.\ \varphi_1$:, this is enumerating elements from the domain, and as such $|\mathcal{M}| \cdot |M|^{|\varphi_1|} \leq |M|^{|\varphi_1|+1}$.
- $\varphi_1 \,\mathcal{S}\, \varphi_2$ and $\varphi_1 \,\mathcal{U}\, \varphi_2$: the worse case is that the number of iterations needed are the length of the whole model which is bounded by j which leads to the following time $|\mathcal{M}|(|M|^{|\varphi_1|} + |M|^{|\varphi_2|}) \leq |\mathcal{M}|^{|\varphi_1|+|\varphi_2|+1}$.
- $\mathcal{K}_b\varphi'$: the worse case is to enumerate all worlds, in which case we have $|\mathcal{M}| \cdot |\mathcal{M}|^{|\varphi'|} \leq |\mathcal{M}|^{|\varphi'|+1}$.

$\square$

3.4 Checking Logs

Next, we focus on how to transform logs into Kripke structures that can be subsequently be checked using Algorithm 3.2. This is an important step for the whole log checking procedure. This is not always easy, as it may be unclear who to assign events to sessions, and the clocks of the different agents may not always be synchronized properly. In the first part of the section, we assume an ideal world, and sketch the steps involved. In the second, we comment on our specific experiences doing so for checking the logs of the 2014 Victoria state election in Australia. Throughout this section we assume that we are given a set of EFOTL signatures, which include the predicate, the agent. However, the worlds are created during the model generation.

3.4.1 Model Generation

Let us assume that we are given the logs generated by all the agents that participated in a distributed system. For simplicity, we assume here that the local clocks of each agent are properly synchronized, for example, accessing a time server, and that all log events are appropriately time stamped. We also assume the events log entries are recorded in plain text. The tasks that we need to execute to reconstruct a Kripke structure are as follows:

- Translate the plain text into pairs of EFOTL predicates and worlds.
- Partition the logs according to the world.
- Map real time into discrete time points that are understood by EFOTL.
- Update the reachability relation according symmetries among sessions.

Translation: For the first step, we consider the log from one agent, and treat other agent's log the same way. We then iterate through the log and translate each entry event into an EFOTL predicate by retrieving a suitable predicate name, and generating a vector of arguments from the domain $\mathbb{D}$. Agent names a will be translated to the corresponding domain element $\ulcorner a \urcorner$ as shown in Example 3.2.2. Then we translate the information in a entry event that identifies the session, in which this particular event was a part, in into a world. Such information could be a *session identifier* or the *serial* *number* of particular ballot. The worlds are global for all logs across the multiple agents after the translation.

Partition: As a next step, we partition each log for each of the agents into sets of events that belong to the same world according to the translation *world identifier*. Such as a log contains information from several sessions, the identifier can be the session id. After this step is complete, we can identify all events that participated in one session, as they share the same world.

Mapping: Next, we can create a total order of all events from all agents in one session, because we assume that they are time-stamped. We assign time points starting with 0 in order to each event. These are the time points for the model. We call this mapping process *synchronization*.

Symmetry: Finally, we update the reachability relation for each agent a and each worlds in the following way. Initially, each world denotes one session. We identify two worlds in the reachability relation R_a, if the sequence of log events for both worlds is equivalent for agent a. This way, we make symmetries for the model checker explicit, who can use this information to optimize search.

After completion of these four steps, we have all the ingrediants to create a Kripke structure for checking our epistemic security policies (see Section 3.2): The Kripke model $\mathcal{M} = (\mathbb{W}, \mathbb{D}, \mu, (R_a)_{a \in \mathbb{A}})$ can now be defined. The worlds $\mathbb{W}$ are defined to be the collection of all sessions, $\mathbb{D}$ the set of all constants that appear in any log, including elements denoting agents. The mapping is defined $\mu(i, w, a)(P)$ as the set of predicate P that are true for agent a, at time point i in world w. Making the symmetry step from above more precise, the family of relations $(R_a)_{a \in \mathbb{A}}$ is defined so each relation is an equivalence relation in the following way: $(w, w') \in R_a$ iff $\forall i P. \mu(i, w, a)(P) = \mu(i, w', a)(P)$.

We illustrate this process using two examples.

Example 3.4.1. An event of the form "@ 2014-10-02-13:00, session 15. Message m sent to principal a" will be translated into "2014-10-02-13:00 (send (m, a), 15)". $\mathbb{D}$ must be chosen to contain elements denoting message m and agent a. "send" is a predicate symbol.

Example 3.4.2. As a concrete example we return to Example 3.2.1 and consider two hypothetical logs depicted in Figure 3.3 generated during three sessions. Each entry of the contains a pair of an EFOTL predicate and the corresponding session (world). The two agents talking to one another are a and b. The setting is such that whenever a sends a message, b will receive

Time	Log from a
13:00:00	$(send(\ulcorner B \urcorner, M), 1)$
13:00:02	$(send(\ulcorner B \urcorner, M'), 2)$
13:00:03	$(send(\ulcorner B \urcorner, M), 3)$

Time	Log from b
13:00:01	$(receive(\ulcorner A \urcorner, M), 1)$
13:00:04	$(receive(\ulcorner A \urcorner, M'), 2)$
13:00:05	$(receive(\ulcorner A \urcorner, M), 3)$

Figure 3.3: Log Sample of three sessions

Time Point	World [1], World[3]		World [2]	
	Log A	Log B	Log A	Log B
0	$send(\ulcorner B \urcorner, M)$		$send(\ulcorner B \urcorner, M')$	
1		$receive(\ulcorner A \urcorner, M)$		$receive(\ulcorner A \urcorner, M')$

Figure 3.4: Result of synchronisation of Log Sample Figure 3.3

it. In the log shown in Figure 3.3, we write directly the predicates *send* and *receive*. Figure 3.4 gives an account for how this information is represented in the Kripke structure. Note, that by symmetries, session 1 and 3 form an equivalence class modulo reachability, whereas 2 remains the only element in the equivalence class generated by 2.

One special thing here is that R_a and R_b are the same here just because their local view of world 1 and world 3 are equivalent, however, if at 13:00:05 b receives M'' instead of M then pair (World 1, World 3) is still in R_a but not in R_b.

3.4.2 Empirical Observations

We now describe our solution to various practical challanges that arose when trying to reconstruct the Kripke structure from the logs for the Victoria 2014 state election.

Translation: We could parse logs line by line, as originally intended. Network errors, printer warning etc. left many multi-line warning together with Java traces in the logs. These exceptions are harmless though form a parsing point of view, as we simply skipped them.

We associate each relevant log entry with a predicate, consisting of a predicate symbol and an array of arguments, i.e. constants in $\mathbb{D}$. In the logs, events are usually represented as large serialized JSON objects, which means that we needed to formulate and execute complex filter operations to access the information relevant for our model.

Partition: One of our main challenges was to identify sessions. Most log

Election	#Agents	#Worlds	Time
Norway Election	2	1000	92sec
Victoria State Election	7	17	1.4sec

Figure 3.5: Experiments

entries, but regrettably not all, provide the serial number of the ballot the event should be associated with. If the serial number exists, we use it as world, if not, we extrapolate a suitable world from contextual information, for example, which was the last serial number seen. This heuristics is very crude, and could be easily improved upon by logging more information.

3.5 Examples of Policy Checking

In this section we demonstrate the usefulness of EFOTL and illustrate its expressiveness by describing a set of epistemic security policies for both, the Norwegian 2013 parliamentary and Victoria 2014 state election. Furthermore, we report on the results of checking the security policies using a prototype implementation of Algorithm 3.2 in Haskell [81]. As input to the model construction, we used in the Victoria case the logs that were given by the Victorian Election Commission, which amounted to 24 ballots. Since we did not have access to the Norwegian logs, we synthesized sample logs for about 1500 ballots, a third of which accounts for double votes. We used those as input to our model-checker.

Below we describe the two case studies in detail. We state the stage, and background and setup of each election system and derive epistemic security policies. Our findings are reported in Figure 3.5.

3.5.1 Norwegian Electronic Voting

Background: Norway offered in 2013 internet voting as a supplement to traditional pen-and paper based voting. 75000 internet ballots were cast. As we did not have access to the logs, we decided to rebuild part of the infrastructure in our lab and focus on the *vote collector* that is the server that collects all incoming encrypted votes and the *cleanser*, i.e. the server that removes double votes. Note, that in a Norwegian election the incoming encrypted votes carry identifiying information, as voters may vote several times. Only the last vote counts.

Time	Vote Collector Server (V)	Cleanser (C)
@ 13:00	$(Ballot(1001), 1)$	
@ 13:02	$(Ballot(1002), 1)$	
@ 13:04	$(Ballot(1003), 2)$	
@ 13:05	$(Ballot(1004), 1)$	
@ 13:06	$(Ballot(1006), 2)$	
@ 14:00		$(Read(1001), 1)$
@ 14:01		$(Read(1002), 1)$
@ 14:03		$(Read(1003), 2)$
@ 14:04		$(Read(1004), 1)$
@ 14:05		$(Read(1006), 2)$
@ 14:07		$(Reject(1001), 1)$
@ 14:08		$(Reject(1002), 1)$
@ 14:09		$(Reject(1003), 2)$
@ 14:11		$(Accept(1004), 1)$
@ 14:12		$(Accept(1006), 2)$

Time point	Voter 1		Voter 2	
	Vote (V)	Cleanser (C)	Vote (V)	Cleanser (C)
0	$Ballot(1001)$		$Ballot(1003)$	
1	$Ballot(1002)$		$Ballot(1006)$	
2	$Ballot(1004)$			$Read(1003)$
3		$Read(1001)$		$Read(1006)$
4		$Read(1002)$		$Reject(1003)$
5		$Read(1004)$		$Accept(1006)$
6		$Reject(1001)$		
7		$Reject(1002)$		
8		$Accept(1004)$		

Figure 3.6: Log Sample for Cleanser and Vote Collector Server before and after synchonisation in Norwegian election.

An example of a log that we analyzed can be found in Figure 3.6. We define the following predicates for reconstructing log events. Note, that each log event is a pair, where the second component represents the voter, which is, in EFOTL, a world.

- $(Ballot(id), v)$ is an event that may occur in the log of the vote collector. It testifies that a ballot was received from voter v with a unique ballot id number id. For simplicity, we omit the ciphertext representing the encrypted vote, as our security policies do not mention it.

- $(\mathit{Read}(\mathit{id}), v)$ is an event that may occur in the cleanser's log and testifies that a ballot with *id* cast by voter v was read.
- $(\mathit{Accept}(\mathit{id}), v)$ is an event that this is the ballot that will count as v's ballot. It also means taht it will be forwarded to the mixer and eventually to the final counting, components, that we have not but could model in our system.
- In contrast, $(\mathit{Reject}(\mathit{id}), v)$ is the log entry that says that this ballot (from voter v) was discarded.

In this case study, there are only two agents, the vote collector V and the cleanser C. Each voter forms a world. As each ballot has its unique identification number, the logs will not exhibit any exploitable symmetries. As a consequence, each $(R_a)_{a\in\{V,C\}}$ will only consists of singelton pairs $\{(v,v)|v \text{ voter}\}$. The result of *synchronisation* can be found in Figure 3.6.

Epistemic Security Policies: The first policy that we describe here is that vote collector and the cleanser agree on which ballots actually exist. This property is expressed by two forumulas that relate *Ballot* events in the log of the vote collector server V and the *Read* events in the log of the cleanser C. The first policy states that each ballot must be read by the cleanser. The second formula states the converse namely that every ballot in the cleanser needs to be in vote collector server.

$$\Box\forall\ \mathit{id}.\ (\mathcal{K}_V(\mathit{Ballot}(\mathit{id})) \to \Diamond\mathcal{K}_C(\mathit{Read}(\mathit{id})))$$

$$\Box\forall\ \mathit{id}.\ (\mathcal{K}_C(\mathit{Read}(\mathit{id})) \to \blacklozenge\mathcal{K}_V(\mathit{Ballot}(\mathit{id})))$$

Furthermore since the order of ballots matters, we wish to make sure that the ordering of any two ballots is preserved between V and C.

$$\begin{aligned}\Box\forall\ \mathit{id},\mathit{id}'.\ (&\mathcal{K}_V(\mathit{Ballot}(\mathit{id}) \wedge \Diamond\mathit{Ballot}(\mathit{id}'))\\ &\to \Diamond\mathcal{K}_C(\mathit{Read}(\mathit{id}) \wedge \Diamond\mathit{Read}(\mathit{id}')))\end{aligned}$$

The remaining policies focus on the cleanser, and establish a relation between the predicates *Reject* and *Accept* on the one side and *Read* on the other. For example, every read ballot must either be accepted or rejected later.

$$\Box\forall\ \mathit{id}.\ \mathcal{K}_C(\mathit{Read}(\mathit{id}) \to \Diamond(\mathit{Accept}(\mathit{id}) \vee \mathit{Reject}(\mathit{id})))$$

A ballot should not both be rejected and accepted.

$$\Box\forall\ \mathit{id}.\ \mathcal{K}_C(\mathit{Accept}(\mathit{id}) \to (\neg\blacklozenge\mathit{Reject}(\mathit{id}) \wedge \neg\Diamond\mathit{Reject}(\mathit{id})))$$

The next policy expresses the only the lastw vote castshould be accepted accepted. The policy requires therefore require that all earlier ballots from the same voter must be rejected. Here we use the $\bigcirc$ modality to make sure we are not counting the same ballot twice.

$$\Box \forall\ id, id'.\ \mathcal{K}_C((\mathit{Read}(id) \wedge \bigcirc \Diamond \mathit{Read}(id')) \rightarrow \Diamond \mathit{Reject}(id))$$

Finally, the last policy guarantees that for every voter who voted, at least one ballot is accepted.

$$\Box \forall\ id.\ \mathcal{K}_C(\mathit{Read}(id) \rightarrow \Diamond \exists\ id'.\mathit{Accept}(id'))$$

Summary: Not surprisingly, our experiments showed that no violations of this set of epistemic security policy were found.

3.5.2 Victoria Sate (Australia) Electronic Voting

The Australian state of Victoria used a variant of the Prêt à Voter [31, 45] cryptographic voting protocol for the 2014 state election. The implementation is called vVote, which uses a centralized logging framework slf4j. While in operation, the system generated detailed logs of the import actions during the election. In this section we describe our case study, where we develop epistemic security policies and checked the logs for violations.

Background: The Victoria State voting system has the following main components: web bulletin board (private and public ones), print-on-demand printer, randomness generation server, electronic ballot maker, etc. The reliability of this system depends on a threshold signature scheme, which allows a subset of the private bulletin board peers above a particular threshold to jointly generate signatures for certain message. Voting in this system has three core procedures, ballot generation, printing ballot on demand and vote casting. All of these procedures are implemented in a distributed way and rely on the interact of the different components of vVote. The logs that we evalute were generated by the private bulletin board peers (MBB), print-on-demand printer (VPS) and the electronic ballot maker (EVM).

Before the election, VPS generates empty ballots with the help of randomness servers. Each ballot contains a permuted candidate list. The process of casting a vote is as follows: The voter enters a polling station, registers, and receives the permuted candidate list before entering the booth. In the booth he or she uses the list to authenticate to the EVM, casts the vote (or alternatively audits the ballot). Upon casting the vote, the voter receives a receipt with his preferences listed in the same order as the permuted candidate list. He or she can check that both match up. As the candidate list is required to be shredded after the voting, the receipt provides evidence that a vote was cast, but it does not reveal to vote preferences.

When a voter begins the vote casting process in the voting booth, a pre-generated ballot is authenticated by MBB and a message *recExt* of type “pod” is logged in MBB’s log and a signed response (*sendRes*) is then sent back to VPS. This is shown in Figure 3.7. Once the signatures of four out of five MBB peers are validated, the VPS logs that the threshold of reporting peers was met (*meThld()*) and it prints out the ballot for the voter.

After the voter obtains the ballot, the voter can decide whether to run a confirmation checking (audit), which ensures the ballot is well-formed, or to cast a vote with this ballot(vote). The audit is done by VPS and reveals the randomness used for permuting the candidates and proves its correctness to MBB with a message of type “audit”. If the voter decides to audit the ballot, he or she may not use it for voting but must instead request a new ballot. A ballot is cast with the help of EVM, and the casting process consists of a start EVM message (with message of type “startevm”) and a vote message (with message of type “vote”). The predicates used are the following:

- $\mathit{sendM}(ty)$: the client sends a message to all MBB peers with a ballot of type ty.
- $\mathit{recExt}(ty)$: MBB peer receives from a client a message with ballot of type ty.
- $\mathit{sendRes}()$: MBB peer sends response with its signature back to the client who sent the external message to it.
- $\mathit{validSig}(\ulcorner pr \urcorner)$: The client validates the signature received from MBB peer pr.
- $\mathit{recRes}(\ulcorner pr \urcorner, ty)$: The client receives the response from peer pr for ballot of type ty.
- $\mathit{metThld}()$: The threshold of responses received and signature validated is met.

Time point	VPS	MBB peer1
0 @ 13:52:36,308	$\mathit{sendM}(94, \text{pod})$	
1 @ 13:52:40,063		$\mathit{recExt}(94, \text{pod})$
		... (Internal Message)
2 @ 13:52:46,759		$\mathit{sendRes}()$
3 @ 13:52:46,762	$\mathit{recRes}(94, \ulcorner\text{peer1}\urcorner, \text{pod})$	
	... (Other peers' Message)	
4 @ 13:52:48,393	$\mathit{validSig}(\ulcorner\text{peer1}\urcorner)$	
5 @ 13:52:48,396	$\mathit{validSig}(\ulcorner\text{peer5}\urcorner)$	
6 @ 13:52:48,398	$\mathit{validSig}(\ulcorner\text{peer3}\urcorner)$	
7 @ 13:52:48,401	$\mathit{validSig}(\ulcorner\text{peer4}\urcorner)$	
8 @ 13:52:48,402	$\mathit{metThld}()$	

Figure 3.7: Logs from VPS, MBB

Epistemic Security Policies: Without loss of generality, we specify our epistemic secruity policies only for MBB peer1. The policies for the other peers are similar, and we check them as well. In this case study, we choose ballot serial numbers to represent worlds. Each such serial number is unique. Wheras the agents are VPS (V), EVM (E) and the five MBB peers (named: peer1 (P_1), peer2 (P_2), ..., peer5 (P_5)). The first policy we show is that when a peer i receives an external messages, they must have been originated from either from VPS or EVM.

$$\begin{aligned}\Box\forall\ ty.\ &\mathcal{K}_{P_i}(\mathit{recExt}(ty)) \rightarrow \\ &\blacklozenge(\mathcal{K}_V(\mathit{sendM}(ty)) \vee \mathcal{K}_E(\mathit{sendM}(ty)))\end{aligned}$$

The second policy that we check is that the threshold is only met if it is proceeded by at least four (out of five) valid signature checks from different peers. Formally we check that at least one of the five possible scenarios are satisfied which is what the formula φ does.

$$\begin{aligned}&\text{Let } \psi(\overline{ps}) &&= \textstyle\bigwedge_{p\in\overline{ps}} \blacklozenge \mathit{validSig}(p)\\ &\text{Let } \mathit{peers} &&= \{\ulcorner \mathit{peer1}\urcorner, \ulcorner \mathit{peer2}\urcorner, \ulcorner \mathit{peer3}\urcorner, \ulcorner \mathit{peer4}\urcorner, \ulcorner \mathit{peer5}\urcorner\}\\ &\text{Let } \varphi &&= \mathit{metThld}() \rightarrow \textstyle\bigvee_{p\in \mathit{peers}} \psi(\mathit{peers}\setminus\{p\})\end{aligned}$$

We check this policy, φ, for every client, i.e for both VPS and EVM.

$$\Box\mathcal{K}_V(\varphi)\ \wedge\ \mathcal{K}_E(\varphi)$$

Third, we check that if a client validates a response from a peer, it should have first received a response from that peer. Since this should hold for every client we define this as formula φ' as before.

$$\text{Let } \varphi' \;=\; \forall\; pr.\; validSig(pr) \rightarrow (\blacklozenge\exists\; ty.\; recRes(pr, ty))$$

We check this policy, φ', for every client VPS and EVM in a way similar as before.

$$\Box\mathcal{K}_V(\varphi') \wedge \mathcal{K}_E(\varphi')$$

Finally, we define two security policy to check two error conditions. The first expresses that any ballot that was audited on peer i, it should not be used later for voting. If such a case occurs EVM should have logged "*error*".

$$\Box\mathcal{K}_{P_i}(recExt(\text{"}audit\text{"}) \;\wedge\; \Diamond recExt(\text{"}startevm\text{"})) \\ \rightarrow \mathcal{K}_E(\Diamond\exists\; pr.recRes(pr, \text{"}error\text{"}))$$

The second being that a voter cannot reuse a ballot, which will also incur a logged error message by EVM.

$$\Box\mathcal{K}_E((sendM(\text{"}startevm\text{"}) \;\wedge\; \bigcirc\Diamond sendM(\text{"}startevm\text{"})) \\ \rightarrow \Diamond\exists\; pr.recRes(pr, \text{"}error\text{"}))$$

Summary We have used the EFOTL checker to check all policies on 17 ballots. Because synchronization failed on some of the logs, we excluded them and hence 7 ballots from our analysis. One of the ballots that we inspected was audited. The result is summarized in Figure 3.5 after the synchronization of peer clocks.

We remark that synchronizing clocks in a distributed environments is challenging. We observe that not all agents synchronized their clocks. As shown in the following example, it is a severe problem for logging. In one session, one agent sends a message to another and logs it at 09:46:15, and the recipient receives the message and logs this event with time stamp 10:45:37. However, the response is logged to have been sent at 10:45:44 and the response is recorded as received at 09:46:25, which shows a clear difference of clocks due to the time zone or other causes. For log checking purpose, we offset the clients time manually to compensate for the difference.

There are two reasons for this synchronization problem. In a distributed system live vvote, time stamps shows the order of events, but the peers of the system are not proper set with the correct time.

The first reason is that the clock of all standalone servers are not properly set and the time zone are not correctly logged, which means the clock is not accurate and the logs missed the time zone information. The second reason is the logged event time is not the exactly time of this event, and this is caused by the time difference between logging and event execution. The first problem is easy to fix using methods network time protocol and logging time zone properly. The second problem can be fixed via logging two entries of the event, one before the execution and one after the execution of the event, and this gives a range of time for the logging checking.

3.6 Metric EFOTL

We continue the work of EFOTL with an extension of the metric property [13] for temporal operators. EFOTL with the metric extension is named as epistemic metric first-order temporal logic (EMFOTL) in the following chapters. The logs from Victoria election is further analyzed as a case study for this extension.

3.6.1

The metric property reasons about a real time interval. For the temporal part of EMFOTL, we use I to refer to a *time interval.* More explicitly, a time interval is defined as $I = [n_1, n_2)$, where n_1 and n_2 are real time clocks, and n_2 is a clock greater than n_1 or $+\infty$. I represents a set $\{i \mid n_1 \leq i \text{ and } i < n_2\}$.

Definition 3.6.1 (Syntax of extended EMFOTL).

$$\begin{aligned}\varphi,\ \psi ::= {} & P_n(t_1, ..., t_{|P_n|}) \mid \neg\ \varphi \mid \varphi\ \wedge\ \psi \mid \exists x.\ \varphi \mid \\ & \bullet_I \varphi \mid \circ_I \varphi \mid \varphi\ \mathcal{S}_I\ \psi \mid \varphi\ \mathcal{U}_I\ \psi \mid \mathcal{K}_a(\varphi)\end{aligned}$$

As in EFOTL, we also use a Kripke structure with a possible world model to describe the semantics of EMFOTL. A Kripke structure is defined as $\mathcal{M} = (\mathbb{W}, \mathbb{D}, \mu, \tau, (R_a)_{a\in\mathbb{A}})$, where $\mathbb{D}$ is the domain, each $R_a \subset \mathbb{W} \times \mathbb{W}$ describe the reachability relation on worlds that belongs to agent a. Recall that different agents may have different and possibly incompatible views on world.

The semantics of EMFOTL for non-temporal part of EMFOTL is the same as we defined before, while the temporal part requires another function. The model is extended by introducing a function τ that maps current world (w)'s time point (i) into a real time value τ_i^w. Time stamps of all

$$
\begin{array}{lll}
(\mathcal{M},\nu,\tau,w,i) \models_a \bullet_I \varphi & \text{iff} & (\mathcal{M},\nu,\tau,w,i-1) \models_a \varphi, \\
 & & \text{and } \tau^w_i - \tau^w_{i-1} \in I \\
(\mathcal{M},\nu,\tau,w,i) \models_a \bigcirc_I \varphi & \text{iff} & (\mathcal{M},\nu,\tau,w,i+1) \models_a \varphi, \\
 & & \text{and } \tau^w_{i+1} - \tau^w_{i} \in I \\
(\mathcal{M},\nu,\tau,w,i) \models_a \varphi\ \mathcal{S}_I\ \psi & \text{iff} & \text{for some } j \leq i,\ (\mathcal{M},\nu,\tau,w,j) \models_a \psi \\
 & & \text{and } \tau^w_i - \tau^w_j \in I, \\
 & & \text{and for all } k \in [j+1, i+1) \\
 & & (\mathcal{M},\nu,\tau,w,k) \models_a \varphi \\
(\mathcal{M},\nu,\tau,w,i) \models_a \varphi\ \mathcal{U}_I\ \psi & \text{iff} & \text{for some } j \geq i\ (\mathcal{M},\nu,\tau,w,j) \models_a \psi \\
 & & \text{and } \tau^w_j - \tau^w_i\ \in I, \\
 & & \text{and for all } k \in [i, j) \\
 & & (\mathcal{M},\nu,\tau,w,k) \models_a \varphi
\end{array}
$$

Figure 3.8: Semantics of EFOTL

agents are synchronized when constructed the model, so τ is agent independent.

There are two ways to look at the temporal operators without metric interval, one is consider it a different operator, and the other is set the interval to be $[0, +\infty)$. Here we use the second option, and omit the $[0, +\infty)$ when used in formula.

A formula φ is bounded if all occurrences of $\mathcal{U}_{[n_1,n_2)}$ in φ, n_2 is not infinity.

Definition 3.6.2 (Semantics of EMFOTL)**.** The meaning of formula φ is defined as $(\mathcal{M},\nu,\tau,w,i) \models_a \varphi$ in Figure 3.8.

3.6.2 Finite Model Checking

Definition 3.6.3 (finite models)**.** A model $\mathcal{M} = (\mathbb{W},\ \mathbb{D},\ \mu,\ \tau,\ (R_a)_{a\in\mathbb{A}})$ is finite iff $\mathbb{W}$ and $\mathbb{D}$ are finite sets and there exists a time point i such that for all $j > i$ or $j < 0$, agent a, world w and predicate symbol P then $\mathcal{M}^j_{(w,a)}(P) = \emptyset$.

The world relation now changes to $(w, w') \in R_a$ if for all time point i, predicate P, $\mu(w,i,a)(P) = \mu(w',i,a)(P)$ and $\tau(w,i) = \tau(w,i)$.

The finite models are defined similarly as for EFOTL, however, the constant point does not apply for EMFOTL structure due to the introducing of time intervals. Because the time interval constrain is satisfied differently, when time point index moves.

So instead of using the constant point for boundary point checking, we here simply define the checking out of boundary exception, because there's no time point defined out side the boundary. Whenever the time point out of the finite model are accessed, the checking algorithm will return *Unkown*. This will have some side effects as when *true* is checked out the scope the algorithm will still return *Unkown*.

3.6.3 Checking Logs

In the Australian voting system technical report [45], there's a requirement states that *the printed empty ballot can only be used within 5 minutes after printed.* We are now able to express this policy with the metric extension. Here shows two formulas that expresses the policy.

$$\Box\mathcal{K}_P(SendRes() \wedge \Diamond_{[300,+\infty)}RecExt(\text{"}startevm\text{"}) \rightarrow \neg\Diamond_{[300,+\infty)}RecExt(\text{"}vote\text{"}))$$

This formalization means if the "startevm" is received by the peer 300 seconds after the ballot is confirmed by the peer, then it can not be used by EVM for casting a vote.

$$\Box\mathcal{K}_P(RecExt(\text{"}vote\text{"}) \rightarrow \blacklozenge(RecExt(\text{"}startevm\text{"}) \wedge \Diamond_{[0,300)}SendRes()))$$

This formalization states if a vote is cast, then there must be a "startevm" for checking the ballot signature and the "startevm" is no later than 300 seconds after the ballot is confirmed by the peer.

When our prototype checking tool is tested, a violation of the 5 minutes police is found in the log entry, in which a ballot generated at "11:08:28" was cast at "11:14:09". An explanation of this problem might be the time problem mentioned in the EFOTL part.

3.7 Conclusion

In this chapter we describe how to verify security policies of distributed systems by inspecting the content of logs of different agents. We describe the logic EFOTL as a security policy language and discuss its properties.

EFOTL is an extension of first-order linear-time temporal logic that allows us to express security policies that refer explicitly to the local knowledge (aka. logs) of agents using epistemic connectives. We describe a model checker algorithm that we applied to checking various epistemic security policies of voting systems. Furthermore we describe a procedure for extracting Kripke structures from logs. As a case study, we showed how to formulate some policies for the Norwegian system, and we also apply our tools to the logs that were generated during the Victoria 2014 state election. Several epistemic security policies are given and checked. In the end, a brief discussion of adding metric time into EFOTL is discussed.

The underlying possible world semantics allows us to identify sessions with worlds. This means, that references to session and time may remain implicit, which leads to elegant formulations of security policies.

Chapter 4

Verifying Voting Schemes[1]

4.1 Introduction

The goal of any social choice function is to compute an "optimal" choice from a given set of preferences. Voting schemes in elections are a prime example of such choice functions as they compute a seat distribution from a set of preferences recorded on ballots. By *voting scheme* we refer to the method for counting ballots and computing who won – as opposed to an actual computer implementation of such a scheme or a scheme describing the process of casting votes via computer. The difficulty in designing preferential voting schemes is that the optimisation criteria are not only multi-dimensional, but multi-dimensional on more than one level. On one level, we want to satisfy each voter, so each voter is a dimension. On a higher level, there are desirable global criteria such as "majority rule" and "minority protection" that are at least partly inconsistent with each other. It is well-known that "optimising" such theoretical voting schemes along one dimension may cause them to become "sub-optimal" along another.

This observation is not new and voting specialists have proposed a series of mathematical criteria [26] that can be used to compare various voting schemes with one another. A classic example is the notion of a Condorcet winner, defined as the candidate who wins against *each* other candidate in a one-on-one contest. Such a winner exists provided that there is no cycle in the one-to-one contest relation. A voting scheme is said to satisfy the Condorcet criterion if the Condorcet winner is guaranteed to be elected when such a winner exists. Another is the *monotonicity criterion* which

[1]Based on "Verifying Voting Schemes" [19], joint work with Bernhard Beckert, Rajeev Goré, Carsten Schürmann and Thorsten Bormer, published in JISA 2014

requires that a candidate who wins a contest will also win if the ballots were changed uniformly to rank that candidate higher.

In practice, theoretical voting schemes are often simplified in many ways when used in real-world elections, typically to reduce their complexity to allow counting by hand. Such practical schemes may not satisfy general properties such as the Condorcet criterion simply because it is intractable to compute the Condorcet winner by hand, but they may satisfy some weaker version of "optimality" that is specific to that particular scheme. It may even happen that one among the optimal winners is chosen at random [25] (as allowed by the Australian Capital Territory's Hare-Clark Method) or that someone other than the optimal winner is elected.

Voting schemes also evolve over time – for national elections in the large, and local elections, union elections, share holder elections, and board of trustee elections in the small. Incremental changes to the electoral system, the tallying process and the related algorithms challenge the common understanding about what the voting scheme actually does. For example, since 1969 some local elections in New Zealand adopted Meeks' method [72], which is a voting scheme for preferential voting that uses fractional weightings in its computations and is too complex to count by hand. This also required an adjustment of understanding about who will now be elected. In general, it is often not clear whether changes to the electoral system improve or worsen the overall quality of a voting scheme with regard to the various dimensions of optimisation. Changes to the electoral system in Germany, for example, have created paradoxical situations where more votes for a party translate into fewer seats and fewer votes into more seats, and have prompted Germany's Supreme Court to intervene at several occasions (see, e.g., [32]).

Many jurisdictions around the world are now using computers to count ballots according to traditional voting schemes. Using computers to count ballots opens up the possibility to use voting schemes which really are optimised along multiple dimensions, while retaining global *desiderata* such as the Condorcet criterion. The inherent complexity of counting ballots according to such schemes means that it may no longer be possible to "verify" the result by hand-counting, even when the number of ballots is small. It is therefore important to imbue these schemes with the trust accorded to existing schemes. Note that our focus is on trust in the voting scheme, not trust in the computer-based process for casting votes.

One way to engender trust in such complex yet "fairer" voting schemes is to specify the *desiderata* when the scheme is being designed, and then formally check that the scheme meets these criteria before proposing changes to the legislation to enact the scheme. Such formal analyses could contribute significant unbiased information into the political discussions that typically involve such legislative changes and also assure voters that the changes will not create paradoxical situations as described above.

Formal analysis, however, is only practicable when we possess formal specifications of the voting scheme. We argue that it is important to give declarative specifications of the properties of a voting scheme for two reasons: (1) For understanding their properties and how they change during the evolution process, so that improving a scheme in one aspect does not by accident introduce flaws with respect to other aspects. (2) For checking the correctness of the scheme from both an algorithmic and implementation perspective. We also argue that general criteria are not sufficient and criteria are needed that are tailor-made for specific (classes of) voting schemes.

The properties in question are difficult to state, to formalise, to understand, to analyse, and to describe declaratively (as opposed to algorithmically) because: the final voting scheme may have to compromise between the conflicting demands of multiple individual desirable properties; the voting scheme may evolve and we may have to revisit these desiderata; even when the properties can be made mathematically precise, the resulting mathematical statement cannot serve as a specification if the electoral law defines a voting scheme that does not (always) compute the optimal solution.

In this chapter, we show that seemingly innocuous revisions to a voting scheme can have serious implications on the desired properties of the system and how analysis techniques employing Satisfiability Modulo Theories (SMT) solvers [24] can be used to discover them. As a running example, we use the preferential voting schemes single transferable vote (STV) that is used in elections world-widely, such as the Victoria election in Australia, but also for smaller professional elections.

In Section 4.3, we define two tailor-made criteria to establish the desired properties of the voting scheme. Both criteria are formulated using first-order logic and are amenable for bounded model checking with Z3, which is the tool of choice for our formal analysis (Section 4.4). Besides the experiments, we also discuss advantages and disadvantages of different verification techniques. Subsequently, we discuss (Section 4.5) a particularly interesting variant of the Single Transferrable Vote Algorithm (CADE-STV) for the board of trustees of the International Conference on Automated Deduction (CADE). We explain its oddities and differences to standard STV, and give a historical account of the conception and the stepwise refinement of the algorithm.

4.2 Basic Definitions

The basic notions related to voting schemes are defined below.

Definition 4.2.1. (Voting scheme, ballot, ballot box, election result) Given a non-empty set $\mathcal{C}$ of candidates, a voting scheme $\langle \mathcal{B}, \mathcal{T} \rangle$ is characterized by:

- a set $\mathcal{B}$ of possible ballots that can be cast by voters;
- a tallying function $\mathcal{T}$ assigning to each possible ballot box an election result, where a ballot box is a (finite) multiset of ballots and an election result is a (finite) duplicate-free sequence of candidates (the elected candidates).

Given a non-empty set $\mathcal{V}$ of voters we assume that each voter casts exactly one ballot in the election, allowing us to use the term voter and ballot interchangeably when identifying specific voters and ballots.

According to our definition of a voting schemes (Definition 4.2.1), the order of ballots in a ballot box is irrelevant and tallying functions are deterministic. Also, by definition, the order in which candidates are elected is part of the election result. These assumptions hold for all voting schemes considered in this chapter, but the results and methods presented in the following apply as well to more general notions of voting schemes.

In this chapter, we focus on preferential voting schemes.

Definition 4.2.2. (Preferential voting scheme) In a preferential voting scheme $\langle \mathcal{B}, \mathcal{T} \rangle$, the possible ballots $b \in \mathcal{B}$ are partial linear orders on candidates.

A ballot for a preferential voting scheme orders candidates according to the voter's preference.

Suppose that `C` candidates, numbered $1, 2, \ldots, \texttt{C}$, are competing. Then, we use the notation $[c_1, c_2, \ldots, c_k]$ for a ballot that ranks a subset of the candidates in decreasing order of preference, where $c_i \in \{1, 2, \ldots, C\}$ and $c_i \neq c_j$ for $i \neq j$.

4.3 Semantic Criteria for Analysing Voting Schemes

We distinguish between general criteria that preferably each voting scheme should satisfy and tailor-made criteria that distinguish between different classes of schemes and capture the essence of particular classes. Both kinds of criteria are important for the specification and analysis of voting schemes.

Below, we first describe a few important examples of general criteria found in literature. We then give two examples for tailor-made criteria applicable to the STV family of voting schemes.

4.3.1 General Criteria

Many general criteria that voting schemes preferably should satisfy have been proposed (for an overview see [26]). Note that, even though these basic criteria seem obvious and indispensable for voting schemes on first sight, they are in fact not always satisfied by each reasonable voting scheme. Most real-word voting schemes violate at least some basic criteria for some possible ballot box input.

An "obvious" and widely used criterion is the *majority criterion*, which states that, if a candidate c is ranked first by a majority of voters, then c must be elected. This is indeed satisfied by all reasonable preferential voting schemes that use votes ranking candidates. However, the majority criterion can be violated by preferential voting schemes where voters can attach a numerical preference to candidates instead of just ranking them (Borda count scheme).

Another "obvious" criterion is the *monotonicity criterion* [114]. Assume that there are two ballot boxes b and b' where b' results from b by raising the preference for a candidate c in one or more of the votes and leaving the votes otherwise unchanged (i.e., a vote of the form $[c_1, \dots, c_{i-1}, c, c_{i+1}, \dots, c_k]$ is replaced by $[c_1, \dots, c_{j-1}, c, c_j, \dots, c_{i-1}, c_{i+1}, \dots, c_k]$ $(j < i)$. The monotonicity criterion states that, if c is elected using the ballot box b, then c must also be elected using b'. Surprisingly, some real-world voting schemes – including STV – do not satisfy monotonicity [114].

A third simple criterion is the *fill-all-seats criterion*, which states that all available seats are filled provided that there are sufficiently many candidates, i.e., $\mathtt{C} \geq \mathtt{S}$. In practice, this criterion is often used in a restricted form, e.g., candidates can be elected only if they reach a certain minimal quota.

The majority criterion fully describes the election result for the simple case of a single seat and a candidate with a majority of first preferences. But we desire criteria characterising the "right" result in increasingly complex situations.

An example is the *Condorcet criterion.* A candidate c is a Condorcet winner if c wins a one-to-one comparison against all other candidates, i.e., for all $c' \neq c$ there are more voters preferring c over c' than there are voters preferring c' over c. The Condorcet criterion states that a Condorcet winner c must be elected if there is one. And, as long as there are open seats and there are Condorcet winners among the remaining candidates, these must also be elected.

The majority and the Condorcet criteria present each an example of a *conditional criterion.* They apply to a ballot box only in the case the given condition (e.g., the existence of a Condorcet winner in the Condorcet Criterion) is satisfied; in this case attesting the property (e.g., a Condorcet winner must be elected). Otherwise they hold trivially. This means, that there are two degenerate cases of a conditional criterion: the first is when the condition is never satisfied by a ballot box, in which case the criterion will always hold, no matter which ballot boxes we apply it to. For the second case, when the property of the criterion simply yields true, the condition of the criterion becomes irrelevant and a similar observation applies. We therefore propose to analyze criteria according to *coverage* and *restrictiveness.*

Coverage Should apply to as many different ballot boxes as possible.

Restrictiveness The number of possible election results for ballot boxes to which the criterion applies should be as restricted as possible.

Returning to our criteria, we note that the majority and Condorcet criteria are very restrictive (they specify exactly one winner), but they do not have good coverage (they only apply if there is a clear winner). The fill-all-seats criterion, on the other hand, has full coverage (it restricts the possible outcome for all ballot boxes), but it is not very restrictive. But, even criteria with poor coverage, such as the Condorcet criterion, provide important insights into voting schemes: It is well known, for example, that STV (which we use as a case study) does not satisfy the Condorcet criterion.

Ideally, one would like to characterise schemes by a criterion that allows exactly one result for every possible ballot box, i.e., has full coverage and is fully restrictive. But for many voting schemes used in practice, such criteria do not exist. In these cases, we rely on tailor-made criteria that strike a compromise between coverage and restrictiveness.

In summary, voting schemes can be analyzed according to many different criteria, some criteria are considered important others less important. We remark, however, that no voting scheme exists that would satisfy all reasonable general criteria simultaneously. In the case of preferential voting, Arrow's impossibility theorem [6] states that no scheme can be designed to satisfy the three fairness criteria:

Unanimity. If all voters prefer candidate A over candidate B, then A is ranked over B in the election result.

Independence of irrelevant options. If some voters change their ballot but keep the relative position of candidates A and B in their ballot, then the relative position of A and B remains unchanged in the election result.

Non-dictatorship. There is no single voter whose preferences always prevail in the election result.

4.3.2 Tailor-made Criteria for Preferential Voting Schemes

As stated previously, many more voting scheme criteria have been developed and are described in the literature. So, as a first approach to specifying and analysing a particular voting scheme, one could select some of these to characterise the scheme's properties. For a detailed analysis, however, that is not sufficient. General criteria cannot distinguish between variants of the same voting scheme (or the number of available general criteria would have to be very high).

For example, for our analysis of preferential voting, we have devised two tailor-made criteria that capture the essence of *preferential* voting (Criterion 2) with *proportional representation* (Criterion 1) and are applicable to our case study STV:

(1) There must be enough votes for each elected candidate.

(2) If the preferences of *all* voters with respect to two particular candidates are consistent, then that collective preference is not contradicted by the election result.

The first criterion only considers number of votes and ignores preferences, while the second criterion only considers preferences and ignores number of votes. This separation of the two dimensions (number of votes and preferences) is the key to finding strong criteria that can be described declaratively.

The two criteria compromise in different ways on the two goals of generality and restrictiveness: Criterion 1 has full coverage. It applies to all ballot-boxes without being too restrictive (as the order of preferences is not considered). Criterion 2 has lower coverage. It only applies if the voters' preferences are not contradictory. In that case, however, it is rather restrictive as only a small number of election results are permissible.

Criterion 2 is a weaker version of the Condorcet criterion that, in contrast to Condorcet, is satisfied by STV. It assumes a preference to be collective if *all* voters agree (or at least not disagree), while the Condorcet criterion assumes a preference to be collective if it is supported by a *majority* of voters.

These two criteria may not cover all the desired properties that we expect to hold, however, they offer a good starting point for a formal analysis of voting schemes, especially those based on STV. For additional properties currently not covered, the two criteria may need to be refined accordingly.

4.3.3 Criterion 1: Enough Votes for each Elected Candidate

One core element of any STV system is the transfer of surpluses from elected to remaining candidates. Here, a surplus is defined by the number of votes of the elected candidate above the required quota. We note, that in some variants of STV, surplus votes are transfered as a whole, whereas other variants of STV transfer votes at a fractional value. To argue that a particular transfer is sensible and justifiable, some regulations require that each ballot can only be used once to elect a candidate marked on the ballot, which leads us to the definition of the first criterion. It says, that the entire ballot box can be partitioned into (disjoint) groups of (used) ballots such that each elected candidate is supported by exactly one group.

Definition 4.3.1. (Criterion 1: Enough votes for each elected candidate) Let $\langle \mathcal{B}, \mathcal{T} \rangle$ be a preferential voting scheme (Def. 4.2.2). Let `Q` be the quota, `C` the number of candidates, `V` the number of voters, and `S` the number of seats.

We define that $\langle \mathcal{B}, \mathcal{T} \rangle$ satisfies Criterion 1 iff, for all ballot boxes

$$\mathscr{b} = \{\!|\beta_1, \ldots, \beta_{\mathtt{V}}|\!\} \text{ with } \beta_i = [c_1^i, \ldots, c_{k_i}^i] \in \mathcal{B}$$

and corresponding election results $\mathcal{T}(\mathscr{b}) = [r_1, \ldots, r_\sigma]$ ($\sigma \leq \mathtt{S}$ is the number of elected candidates), there is a partition

$$\mathscr{b} = \mathscr{b}_1 \mathbin{\dot\cup} \ldots \mathbin{\dot\cup} \mathscr{b}_\sigma \mathbin{\dot\cup} \mathscr{b}_{rest}$$

such that, for $1 \leq i \leq \sigma$, the following holds:

1. $|\mathscr{b}_i| = \mathtt{Q}$ (there are exactly $\mathtt{Q}$ votes in each class that supports an elected candidate).

2. $r_i \in \beta$ for all $\beta \in \mathscr{b}_i$ (each vote β in the class $\mathscr{b}_i$ supports candidate r_i, i.e., the candidate occurs somewhere among the preferences of β).

Note, that here the actual order of preferences is not taken into consideration.

Example 4.3.1. Assume there are four candidates A, B, C, D for two vacant seats, the votes to be counted are $[A, B, D], [A, B, D], [A, B, D], [D, C], [C, D]$, and the quota is $\mathtt{Q} = 2$. The election result $[A, D]$ satisfies Criterion 1 using the partition $\{[A, B, D], [A, B, D]\}$, $\{[C, D], [D, C]\}$, $\{[A, B, D]\}$, where the first group supports candidate A and the second supports candidate D.

Example 4.3.2. Criterion 1 does not capture election results exactly, it over-approximates them: Since we ignore the order of preference markings, the criterion may hold for unintended election results. The result $[B, D]$ is not a valid election result because it violates the majority criterion: A, despite its majority of first preferences, is not elected. Nevertheless, it satisfies Criterion 1, by virtue of the same partition from the previous example, because the first group also supports B as elected candidate.

Example 4.3.3. But, Criterion 1 also rules some election results as invalid. The result $[A, B]$, for example, which contradicts proportional representation, is not supported by this or any other partition (which shows that this criterion is indeed related to the requirement of proportional representation).

Formalisation To formalise the criteria, we use first-order logic over the theories of natural numbers and arrays with the following notation in addition to the notation defined previously:

b: is a two-dimensional array representing the ballot box ($\mathcal{b}$ in Definition 4.3.1), where $b[i,j] \in \{1,\ldots,\mathtt{C}\}$ represents the number c_i^j of the candidate that is ranked by vote i in the jth place. Thus, i's preference is $[b[i,1], b[i,2], \ldots]$. If vote i ranks only $k \leq \mathtt{C}$ candidates, then $b[i,j] = 0$ for $k < j \leq \mathtt{C}$.

r: is an array representing the result $\mathcal{T}(\mathcal{b})$, where $r[i]$ is the ith candidate that is elected ($1 \leq i \leq \mathtt{S}$). If less than $\mathtt{S}$ candidates are elected, then $r[i] = 0$ for the empty seats.

Our criterion is formalised by a formula ϕ in which all the above (free) variables occur. We also use an existentially quantified variable a of type array that represents the partition and the assignment of classes in the partition to elected candidates as follows:

$a[i] = k$ if the ith vote supports the kth elected candidate $r[k]$. If the ith vote does not support any elected candidate, then $a[i] = 0$. Using this representation, a class $\mathcal{b}_k$ (as introduced in Definition 4.3.1) can be written as: $\mathcal{b}_k = \{\beta_i \mid a[i] = k, 1 \leq i \leq \mathtt{V}\}$; votes not supporting any candidate are collected in $b_{rest} = \{\beta_i \mid a[i] = 0\}$.

Then, the formula $\phi = \exists a(\phi_1 \wedge \ldots \wedge \phi_4)$ is the existentially quantified conjunction:

$$\forall i\big(1 \leq i \leq \mathtt{V} \rightarrow 0 \leq a[i] \leq \mathtt{S}\big) \qquad (\phi_1)$$

$$\forall i\big(1 \leq i \leq \mathtt{V} \rightarrow (a[i] \neq 0 \rightarrow r[a[i]] \neq 0)\big) \qquad (\phi_2)$$

$$\forall i\big((1 \leq i \leq \mathtt{V} \wedge a[i] \neq 0) \rightarrow \exists j(1 \leq j \leq \mathtt{C} \wedge b[i,j] = r[a[i]])\big) \qquad (\phi_3)$$

$$\begin{aligned} &\forall k\big((1 \leq k \leq \mathtt{S} \wedge r[k] \neq 0) \rightarrow \\ &\quad \exists count(count[0] = 0 \wedge \\ &\qquad\quad \forall i(1 \leq i \leq \mathtt{V} \rightarrow (a[i] = k \rightarrow count[i] = count[i-1] + 1) \wedge \\ &\qquad\qquad\qquad (a[i] \neq k \rightarrow count[i] = count[i-1])) \wedge \\ &\qquad\quad count[\mathtt{V}] = \mathtt{Q})\big) \end{aligned} \qquad (\phi_4)$$

Formulas ϕ_1 and ϕ_2 express well-formedness of the partition. Formula ϕ_3 expresses that only votes can support a candidate in which that candidate is somewhere ranked. Formula ϕ_4 expresses that each class supporting a particular elected candidate has exactly $\mathtt{Q}$ elements. To formalise this, we use an array *count* such that $count[i]$ is the number of supporters among votes $1, \ldots, i$ that support the kth elected candidate.

Note, that this criterion assumes all seats to be filled and has to be relaxed if a voting scheme does not satisfy the fill-all-seats criterion or there are not enough candidates that can reach the quota.

4.3.4 Criterion 2: Election Result Consistent with Preferences

We also consider a second criterion that can be considered orthogonal to Criterion 1. We check that the election result respects the preferences of the majority, as stated in the following definition.

Definition 4.3.2. (Criterion 2: Election result consistent with preferences) Given a ballot box $\mathcal{b}$, let $R = \bigcup \mathcal{b}$ be the union of the partial linear orders given by the votes $\beta \in \mathcal{b}$, and let R^+ be the transitive closure of R. Then, there is an argument for ranking candidate c_1 over c_2 iff $c_1 R^+ c_2$.

We define Criterion 2 to hold for a preferential voting scheme $\langle \mathcal{B}, \mathcal{T} \rangle$ iff, for all ballot boxes $\mathcal{b}$ and for all candidates x, y, the following holds:

> If there is an argument for ranking x over y but not for ranking y over x then y must not be ranked higher than x in the election result $\mathcal{T}(\mathcal{b})$.

Note that, in the above definition, R and R^+ may not be order relations.

Formalisation That there is an argument for ranking x over y implies that there is a sequence c of candidates such that $x = c[0], \ldots, c[k] = y$ and there is a sequence of votes $v[1], \ldots, v[k]$ such that $v[i]$ prefers $c[i-1]$ over $c[i]$ $(1 \leq i \leq k)$.

We formalise that vote $v[i]$ prefers candidate c_1 over candidate c_2 by:

$$\begin{aligned}\phi(v, i, c_1, c_2) \;=\; & \exists j (1 \leq j \leq \mathtt{C} \wedge b[v[i], j] = c_1 \wedge \\ & \quad \forall j' (1 \leq j' < j \rightarrow b[v[i], j'] \neq c_2))\end{aligned}$$

The first line of the above formula says that voter $v[i]$ gives the preference j to candidate c_1. The second line says that v does not give a higher preference $j' < j$ to c_2: i.e., gives c_2 lower preference or no preference at all.

Now, we can formalise that there is an argument for ranking x over y by:

$$\begin{aligned}\Phi(x,y) \;=\; & \exists v \exists c \exists k (x = c[0] \wedge y = c[k] \wedge \\ & \quad \forall i (1 \leq i \leq k \rightarrow (1 \leq v[i] \leq \mathtt{V} \wedge 1 \leq c[i] \leq \mathtt{C} \wedge \\ & \qquad \phi(v, i, c[i-1], c[i]))))\end{aligned}$$

In a similar way as with ϕ, we can formalise the fact that the voting result gives a higher ranking to candidate c_1 than to candidate c_2 as follows:

$$\begin{aligned}\psi(c_1, c_2) \;=\; & \exists j (1 \leq j \leq \mathtt{S} \wedge r[j] = c_1 \wedge \\ & \quad \forall j' (1 \leq j' < j \rightarrow r[j'] \neq c_2))\end{aligned}$$

Using the formulas Φ and ψ, the criterion can be formalised as follows:

$$\forall x \forall y \big((1 \leq x \leq \mathtt{C} \wedge 1 \leq y \leq \mathtt{C} \wedge x \neq y \wedge \Phi(x,y) \wedge \neg\Phi(y,x)) \;\rightarrow\; \neg\psi(y,x)\big)$$

4.4 Checking Properties Using SMT Solver

4.4.1 Overview: Different Approaches to Verification with SMT Solvers

In this section we discuss a range of methods that employ Satisfiability Modulo Theories (SMT) solvers for verifying that a voting scheme satisfies any of the aforementioned semantic criteria. Our motivation to chose SMT solvers for this task is twofold: firstly, SMT solvers have evolved into powerful reasoning tools that are successfully used in model checking and software verification, and secondly, the theories supported by SMT solvers allow us to express semantic criteria easily.

Modern SAT solvers are programs that efficiently decide the satisfiability of a given set of formulas of classical propositional logic [53]. Although this problem is NP-complete, modern SAT solvers can easily solve problem instances with hundreds of propositional variables.

SMT solvers are SAT solvers that are extended by theories meaning they provide domain-specific and highly optimised solvers for arithmetic, arrays, uninterpreted functions, and so on. There are many SMT solvers under active development, such as CVC, MathSAT5, Yices, Z3, etc. and there are annual SMT solver competitions continuously driving the progress regarding theoretical and engineering aspects of SMT solver technology. Among all SMT solvers, Z3 has emerged as a powerful and comprehensive tool that is also practical in that it provides APIs for various programming languages, for example Python. Z3's support for first-order logic over the theories of natural numbers and arrays is outstanding, which we applied to formalize semantic criteria of voting systems. These are the main reasons why we chose Z3 to conduct our experiments.

In the following, we first give an overview of the different ways to apply SMT solvers for checking semantic criteria of voting schemes and present experimental results for the applicability of one of these methods. We address the question of how to represent semantic criteria in the input language of Z3 (using Z3's Python API) and demonstrate feasibility of expressing common semantic criteria at several examples.

To choose a particular technology for the task of verification, we must strike a balance between the ease of use of a particular tool and the quality of the resulting proof. In general, it is not possible to combine both full automation and a full verification, which would be the most desirable result.

Full verification is to provide a general correctness argument for arbitrary vote instances of any size. In order to achieve full verification using SMT solvers, we first need to provide the solver with a logical representation of the voting algorithm (the implementation of the voting scheme). These representations can be derived using off-the-shelf methods, such as weakest precondition generation [11]. As these tools usually cannot derive all loop invariants automatically, they may have to be assisted by a manual, time intensive, error-prone, and sometimes unsuccessful process (see Sec. 4.4.3). If, however, the loop invariants are known, SMT solvers can be used to discharge first-order proof obligations,

If we unroll loops in the voting algorithm to a specified finite bound, we speak of *bounded verification*. The advantage over full verification is that, as there are no loops left after unrolling, no manual assistance is required. The disadvantage is that we provide a proof of correctness only for a subset of vote instances.

Perhaps the most automatic but in general least precise method of verification is *bounded model checking*, which exhaustively checks properties for finitely many concrete vote instance up to a certain size. While the scalability of this method is restricted in general, its main strengths are that it neither relies on explicit loop invariants nor weakest precondition generation. It is thus a good candidate for the initial examination of a voting scheme.

4.4.2 Bounded Model Checking

We examine voting schemes for their semantic criteria by exhaustively testing voting instances using a bounded model checker. That is, we exhaustively run the voting algorithm on the fixed set of input data defined by a given bound and test whether the result produced by each concrete execution of the algorithm satisfies the criterion. If the bounded model checker does not find a bad state, we have established that the criteria are satisfied, which by the small scope hypothesis [80] is not a proof but indicates the absence of programming bugs and conceptual problems. If the model checker finds a bad state, it is possible to extract a counter example for future inspection. Bounded model checking is well understood, and its application to voting schemes was discussed in an earlier paper [18], where linear logic was used to express voting schemes, and bounded model checking was performed via proof search within linear logic.

Here we check the criteria using Z3 by encoding the semantic criteria in first-order logic (plus theories supported by Z3) instead of a linear logic framework. In addition we encode input and output of the individual execution of the voting algorithm, as well as relevant intermediate values of program variables. The generated formula is satisfiable iff the semantic criteria hold for the result produced by the voting algorithm. If Z3 reports that the formula is unsatisfiable, the voting instance serves as a counter-example.

We comment on two scalability issues when using exhaustive testing to verify properties of a voting scheme: (a) the size of the input given to a single test run (e.g., the number of ballots or candidates) and (b) the number of different possible ballot boxes (and thus test runs needed) given an upper bound on the number of ballots and candidates. Unsurprisingly, exhaustively testing all voting instances for a large number of votes or candidates is intractable. But even the result of a single test run may be difficult to check if the input is large and we require quantification over array elements of arrays whose size depends, e.g., on the number of votes.

If a criterion does not only relate a single ballot box to the result produced by the voting algorithm, but involves multiple ballot boxes, applicability of exhaustive testing is restricted even further. One example for such a criterion is monotonicity (see Sec. 4.3.1), which relates the ballot box B, the input to the voting algorithm, to another, slightly changed ballot box B'. As a prerequisite to be able to determine whether the semantic criterion holds for a particular voting instance, for all such ballot boxes B'

we need to know the result of applying the voting algorithm to B'. While we can still apply instance checking as before by simply running the checks for vote instance B with all possible ballot boxes B', this clearly aggravates the scalability issues.

One improvement to this "brute force" approach is to narrow down the search space by generating only ballot boxes B' that actually relate to the vote instance B as required by the semantic criterion (e.g., for monotonicity, enumerate only the B' which result from B by solely raising the preference for a candidate c in one or more of the votes). A different solution to this problem is to use weakest precondition generation as described in Sec. 4.4.3 to capture the effect of the voting algorithm in first-order logic with theories.

Furthermore, concerning scalability of instance checking, the performance of the SMT solver depends crucially on the way semantic criteria and relevant data of the program run are encoded in first-order logic over theories. The effects of different encodings on performance are shown in the following.

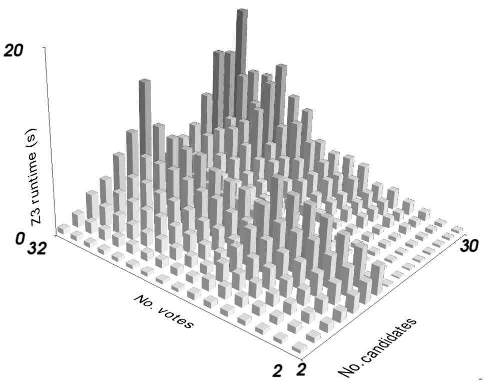
Experiments: Checking Tailor-made Criteria for STV using Z3

To demonstrate that the use of bounded model checking tools is viable, we report here on our experiments on STV implemented in Python using the SMT-based model checker Z3 on the two semantic criteria of STV described in Sections 4.3.3 and 4.3.4.

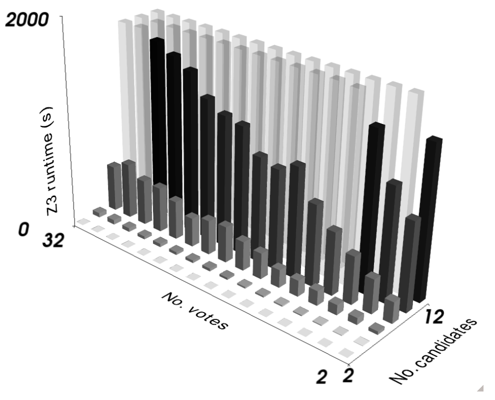
We first produced a straightforward encoding in Z3 of the semantic properties as given in first-order logic. This one-to-one encoding preserves the structure of the original first-order formulas (including all quantifiers) and uses Z3's theories of integers and arrays. Unfortunately, Z3 was not able to handle quantification over the integers in the one-to-one encoding, i.e., Z3 could not determine whether the formula in question was satisfiable or not.

In a second experiment, we eliminated most of the quantifiers in the formula given to Z3 by replacing a universally quantified formula by a conjunction over all instances, and replacing some existentially quantified formulae by providing witnesses (e.g., the array variable a in the first STV criterion), which is possible as the quantifiers range over a small bounded domain.

With quantifiers mostly eliminated, Z3 was able to check the properties for concrete voting instances as shown in Figure 4.1b.



(a) Encoding over bitvectors (no quantifier elimination)



(b) Encoding over integers with quantifier elimination

Figure 4.1: Z3 performance for checking both semantic criteria (as stated in Sec. 4.3.3 and Sec. 4.3.4) of single STV vote instances. The z-axis shows average Z3 run-time in seconds for a single, randomly chosen ballot box with fixed number of candidates and votes. Translucent bars indicate that test runs exceeded a timeout of 1800s. Test runs executed on 12-Core AMD Opteron Processor at 2.1 GHz, with 32 Z3 instances running in parallel.

A third encoding variant takes advantage of the fact that we are in the bounded case, which means that the integers used in a concrete instance are also of bounded size and can thus be represented by bitvectors of a fixed length. For bitvectors, Z3 provides a strategy good enough to handle formulas with the original structure and without eliminating quantifiers. In fact, as shown in Figure 4.1a, for bitvectors, Z3 achieves clearly better performance than reasoning over the integers with quantifiers eliminated. The drop of Z3's runtime at the boundary of 14 candidates respectively votes indicate a change in Z3's proof search strategy – whether this can be further exploited to improve scalability of our model checker is future work.

Combining the use of bitvectors with quantifier elimination does not lead to further increases in performance. This indicates that Z3's internal handling of quantifiers for the bitvector theory is more sophisticated than what is achieved by our elimination of quantifiers in the formula given to Z3.

Regardless of the encoding, the bounded model checking technique does not scale to the size of ballot boxes found in real elections. It is, however, still useful (due to the small model hypothesis) to pinpoint some of the possible errors in the voting scheme, and to check specific larger instances of interest.

Using Z3 and Its Interface

In this section, we give a brief introduction to Z3, and explain how we use Z3 to check encodings of semantic criteria introduced in Sec. 4.3.1.

Z3 provides APIs for C/C++, .Net, Python, and OCaml. In this chapter, we consider only the Python version. The Z3 API comprises many classes and functions, only some of which are explained here (a complete manual can be found on the Z3 website[2]).

At the class level, the `Solver` class plays a central role. It allows to add assertions, check assertions for consistency, and generate models for consistent assertions.

The API provides functions to construct formulas. For example, the `Int()` and `BitVec()` functions create an integer and a bit-vector constant respectively. The functions `Array()` and `K()` create array variables constants respectively. And the functions `ForAll()`, `Exists()`, `And()`, `Or()`, `Not()` provide logical operators.

Generating a Z3 input formula consists of two parts. One is encoding the voting instance from electoral raw data; the other is formalising the criteria.

A voting instance consists of the voting settings (e.g., the number of candidates, the number of voters, the number of vacant seats, etc.), the ballots, the voting result (elected candidates), and sometimes the intermediate variables such as each ballot's support candidate in STV. Entities such as candidates are represented by integers instead of names.

Examples for constructing voting instances in Z3 using bitvectors and integers, respectively, are shown in Figure 4.2. The function `to1DimZ3Array` converts a list of integers into bit-vectors, the function `to2DimZ3Array` is defined analogously and hence omitted.

The integer library provides a function `Update()` that we use set values of voting instances. Alternatively, we can use assignments in formulas that are subsequently executed by the solver.

Next we explain the construction of Z3 input formalising criteria. The majority criterion and the first criterion from Sec 4.3.3 are shown as an example in Figure 4.3 and 4.4, respectively. We explain each in turn.

In Figure 4.3, the definition of `b` (ballot) and `r` (result) are transcribed literally from the formulation of the criterion. In the formula, the universal quantifiers are unfolded into conjunctions (using `And()`), and similarly, existential quantifiers are unfolded into disjunctions (using `Or()`). In line 7,

[2]http://research.microsoft.com/en-us/um/redmond/projects/z3/z3.html

Voting instance with bitvectors

```
def to1DimZ3Array(a):
    AllZero = K(Char, BitVecVal(0, BITVECSIZE))
    res = AllZero
    for i, val in enumerate(a):
        res = Update(res, i + 1, val)
    return res

b = to2DimZ3Array(ballots_instance)
r = to1DimZ3Array(result_instance)
```

Voting instance with bitvectors

Voting instance with integers

```
ballot_sort = ArraySort(IntSort(), IntSort())
b = Array('ballots', IntSort(), ballot_sort)
for i in range(V):
    one_ballot = Array('ballot', IntSort(), IntSort())
    for j in range(C):
        one_ballot = Update(one_ballot, j, ballot_instance[i][j])
    b = Update(b, i, one_ballot)

r = Array('result', IntSort(), IntSort())
for i in range(S+1):
    r = Update(r, i, result_instance[i])
```

Voting instance with integers

Figure 4.2: Formalisation of Voting Instance

Majority criterion

```
[ Implies(Exists(count, // count is a counter
                 And(count[0] == 0, count[V] > V/2, // V is total number of voters
                     And([And(Implies(b[i][0] == c, count[i+1] == count[i]+1),
                              Implies(b[i][0] != c, count[i+1] == count[i]))
                          for i in range(V)]))),
          Or([r[j] == c for j in range(1, S+1)])) // S is the number of elected candidates
  for c in range(1, C+1) ] // C is the total number of candidates
```

Majority criterion

Figure 4.3: Encoding of the majority criterion

`for c in range(1, C+1)` is used to represent "for all candidates". The body of this loop reads as follows: If a candidate accumulates more than 50% of the votes, he or she will appear in the in the final elected result `r`.

Figure 4.4 depicts the formalisation of the first criterion from Sec 4.3.3 in Z3. The formalisation is a one-to-one transliteration. Consider, for ex-

First criterion

```
F1 = [And(a[i] >= 0, a[i] <= S) for i in range(V)]

F2 = [Implies(a[i] != 0, r[a[i]] != 0) for i in range(V)]

F3 = [Implies(a[i] != 0, Or([b[i][j] == r[a[i]] for j in range(C)]))
      for i in range(V)]

F4 = [ForAll(k,
             Implies(And(1 <= k, k <= S, r[k] != 0),
                     Exists(count,
                            And(count[0] == 0,
                                count[V] == Q,
                                And([And(Implies(a[i] == k, count[i+1] == count[i]+1),
                                         Implies(a[i] != k, count[i+1] == count[i]))
                                     for i in range(V)])))))]
```

First criterion

Figure 4.4: Formalisation of First Criteria for STV

ample, the Z3-formula `F1` that encodes the corresponding first order formula

$$\forall i\big(1 \leq i \leq \mathtt{V} \rightarrow 0 \leq a[i] \leq \mathtt{S}\big) \ .$$

Note that there is an index shift. While the voters' index starts from 1 in the original formalisation, it starts from `0` in the Z3 representation.

4.4.3 Full and Bounded Verification

Scalability issues of the bounded model checking approach, which requires exhaustive instance checking, call for other techniques that allow to analyze either all voting instances up to a considerable size, or even guarantee correctness independent of the size or concrete content of the ballot box. In the following, we demonstrate two established methods for this purpose on a small sample program.

For this, consider the program to compute the sum of the first n integers plus a constant c (both c and n are input parameters), as shown in Figure 4.5a. Besides the actual program (in black), we label (in gray) properties of the program state during program execution. The property $n \geq 0$ is the precondition of the program and $x = (n^2 + n)/2 + c$ its postcondition. The precondition captures in which states we intend this program to be invoked, and the postcondition states what property to expect after the program terminates.

$n \geq 0$

```
int i = 0
int x = 0
while (i < n)
{
  i++
  x += i
}
x += c
```

$x = \frac{n^2 + n}{2} + c$

(a) Original program

$n \geq 0$

```
int i = 0
int x = 0
while (i < n)
inv i ≤ n
∧ x = (i^2 + i)/2
{
  i++
  x += i
}
x += c
```

$x = \frac{n^2 + n}{2} + c$

(b) Program with annotated loop invariant

$n \geq 0$

```
int i = 0
int x = 0
if (i < n) {
  i++
  x += i
  if (i < n) {
    assume false
  } else goto out
}
out: x += c
```

$x = \frac{n^2 + n}{2} + c$

(c) Program with loops unrolled once

Figure 4.5: Verification example

With full verification, we are able to prove that the program adheres to its pre- and postcondition pair (using weakest precondition computation, as explained below). For the unbounded case, the user needs to supply information describing the effect of executing (a variable number of) loop iterations in the program via loop invariants. For our example program, the appropriate invariant is shown in Figure 4.5b. Finding the right loop invariant for this example is trivial – for concrete voting algorithms, however, this is a difficult and sometimes infeasible task, as there might not even be a suitable abstraction in form of a concise invariant that describes the loop's effects.

Another option which does not need further user-supplied information is to use bounded verification. Loops in the program are dealt with by examining only program executions up a small number of loop iterations (the *bound*). This allows us to transform loops into if-cascades – as depicted in Figure 4.5c for executions containing at most one loop iteration. The assume statement inserted by the transformation in the second if-block is used for weakest precondition computation: independently from the properties that actually hold at this point in the execution, this statement adds *false* as an assumption so that from this point on, *any* property holds, effectively treating all executions of the original program that pass this point as if they unconditionally establish the postcondition.

Starting from either the original program with annotated invariants or

```
   $(0 < n \rightarrow \big((1 < n \rightarrow \text{true}) \;\wedge (1 \geq n \rightarrow 1 + \text{c} < 9)\big) \;\wedge$
   $(0 \geq n \rightarrow 0 + \text{c} < 9)$
   int i = 0; int x = 0
   $(i < n \rightarrow \big((i+1 < n \rightarrow \text{true})\wedge$
5              $(i+1 \geq n \rightarrow \text{x} + (i+1) + \text{c} < 9)\big) \;\wedge$
   $(i \geq n \rightarrow \text{x} + \text{c} < 9)$
   if (i < n) {
     $(i+1 < n \rightarrow \text{true}) \wedge (i+1 \geq n \rightarrow \text{x} + i + \text{c} < 9)$
     x += i; i++
10   $(i < n \rightarrow \text{true}) \wedge (i \geq n \rightarrow \text{x} + \text{c} < 9)$
     if (i < n) {
       $\text{false} \rightarrow \text{x} + \text{c} < 9 \Leftrightarrow \text{true}$
       assume $\text{false}$
       $\text{x} + \text{c} < 9$
15   } else goto out
   }
   $\text{x} + \text{c} < 9$
   out: x += c
   $\text{x} < 9$
```

Figure 4.6: Weakest precondition computation for property: $x < 9$

the unrolled program, the next step in showing correctness of the program is to generate the weakest precondition for the given postcondition with respect to the program, resulting in a first-order logic formula. The weakest precondition is a property of program states s.t., if the program is started in a state satisfying the weakest precondition, then it terminates in a state that satisfies the postcondition. If the actual precondition of the program implies the weakest precondition, then the program is correct with respect to the given pre-/postcondition pair. Whether this implication holds is typically checked automatically using an SMT solver.

For our example program, Figure 4.6 shows the intermediate properties resulting from weakest precondition computation for a simple postcondition P: $x < 9$. For P to hold after execution of the final statement `x+=c` in line 18 of the program, P with all occurrences of x replaced by $x + c$ has to be true beforehand. Corresponding rules for the other statement types are applied to all statements of the program to get the weakest precondition of the program with respect to P, as seen in lines 1-2 in Figure 4.6. The constructed weakest precondition is equivalent to $(n = 1 \rightarrow c < 8) \wedge (n \leq 0 \rightarrow c < 9)$.

As explained, bounded verification only guarantees correctness up to a given number of loop iterations. Thus for parameters that affect the number of loop iterations in the program execution (in our example: n), we only get correctness results for a subset of values. In the weakest precondition for our example program in Figure 4.6, the sub-formula ($1 < n \to \text{true}$) captures this: for any concrete value for n greater than one, the whole formula simply evaluates to true, although the postcondition $x < 9$ actually may not hold.

With the number of loop iterations depending on the values of input parameters, we get similar scalability issues as described for bounded model checking, if we want to examine larger parameter values. In contrast to bounded model checking, however, for variables not affecting the number of loop iterations, the result of bounded verification applies to *all* values, as variables are handled symbolically by the technique. Additionally, on this abstract level, analysis of program properties might be simplified by exploiting symmetries in the program behaviour. For these reasons, bounded verification is a promising approach to check properties of voting schemes.

4.5 Case Study: Variants of the STV Scheme

Single transferable vote (STV) is a preferential voting scheme [115] for multi-member constituencies aiming to achieve proportional representation according to the voters' preferences.

4.5.1 The Standard Version of STV

There are many versions of STV, but most are an extension or variant of the standard version that is shown in Figure 4.7.

For input and output of the algorithm, we use the same notation and encoding as in Section 4.3. There are `V` voters electing `S` of `C` candidates, and:

b: is the input ballot box, where $b[i, j]$ is the number of the candidate that is ranked by vote i in the jth place. If the vote does not rank all candidates, then $b[i, j] = 0$ for the empty places.

r: is the output election result, where $r[i]$ is the ith candidate that is elected ($1 \leq i \leq \mathtt{S}$). If less than `S` candidates are elected, then $r[i] = 0$ for the empty seats.

We assume the input for the algorithm to satisfy the following conditions (which are pre-conditions for running the standard STV algorithm): (1) $\mathtt{C} \geq$

Standard Version of STV

```
// Initialisation
r  := $[0, ..., 0]$;           // no one elected yet
e  := 1;                        // e is the next seat to be filled
cc := C;                        // cc is the number of (continuing) candidates
Q  := $\lfloor V/(S+1) \rfloor + 1$;     // Droop quota

// Main loop: While not all seats filled and
//            there are more continuing candidates than open seats
// In each iteration one candidate is elected or one candidate eliminated
while $(e \leq S) \wedge (cc > S - e + 1)$ do
    // QuotaReached is the set of candidates for which the number of
    // first-preference votes reaches or exceeds the quota Q
    QuotaReached := $\{c \mid 1 \leq c \leq C \wedge \#\{v \mid 1 \leq v \leq V \wedge b[v,1] = c\} \geq Q\}$;
    if QuotaReached $= \emptyset$ then
        // no one has reached the quota,
        // eliminate a weakest candidate by deletion from the ballot box
        Weakest := $\{c \mid 1 \leq c \leq C \wedge \#\{v \mid 1 \leq v \leq V \wedge b[v,1] = c\}$ is minimal$\}$;
        choose $c \in$ Weakest;
        delete(c);
    else
        // one or more candidates have reached the quota,
        // elect one of them
        choose $c \in$ QuotaReached;
        r[e] := c;  // put c in the next free seat
        e := e + 1; // increase the number e of the next seat to be filled
        do Q times                                  // Q of the votes that
            choose $i \in \{i \mid 1 \leq i \leq V \wedge b[i,1] = c\}$;  // give c top preference
            for j = 1 to C do $b[i,j]$ := 0; od      // get erased
        od
        delete(c);  // delete c from the ballot box
    fi
    cc := cc - 1;    // in any case we have one less continuing candidate
od

// Fill the empty seats
if $e < S$ then
    fill the remaining seats $r[e, \ldots, S]$ with the remaining cc candidates

// procedure for deleting candidate c from votes in b
procedure delete(c) begin
    for i = 1 to V do for j = 1 to C do
        if $b[i,j] = c$ then
            for k = j to $C - 1$ do $b[i,k]$ := $b[i,k+1]$ od;
            $b[i,C]$ := 0;
        fi
    od od
end
```

Standard Version of STV

Figure 4.7: The standard STV algorithm

S, (2) $\mathtt{V} \geq 1$. and (3) votes are linear orders of a subset of the candidates, i.e., for all $1 \leq i \leq \mathtt{V}$ and all $1 \leq j, j' \leq \mathtt{C}$:

- $0 \leq b[i, j] \leq \mathtt{C}$,
- if $b[i, j] \neq 0$ and $j \neq j'$ then $b[i, j] \neq b[i, j']$,
- if $b[i, j] = 0$ then $b[i, j'] = 0$ for all $j' \geq j$.

The initialisation part of the STV algorithm, in particular, computes a quota necessary to obtain a seat (line 5). Different definitions of quotas are used in practice. The most common is the Droop quota $\mathtt{Q} = \lfloor \mathtt{V}/(\mathtt{S}+1) \rfloor + 1$.

To determine the election result, STV uses an iterative process, which repeats the following two steps until either a winner is found for every seat or the number of remaining candidates equals the number of open seats (lines 10–33).

1. If no candidate reaches the quota of first-preference votes, a candidate with a minimal number of first-preference votes is eliminated and that candidate is deleted from all ballots (lines 17–19).

2. Otherwise one of the candidates with Q or more first-preference votes is chosen (line 23) and declared elected (line 24). Of the first-preference votes for that candidate, Q are chosen and erased (lines 26–29). These are the votes that are considered to have been "used up". If the candidate has more than Q votes, the surplus votes remain in the ballot box. Finally, the elected candidate is deleted from all ballots still in the box.

The procedure for deleting a candidate c (lines 40–47) works by searching for the candidate in each vote and, if c is found to have preference j, then the candidate with preference $j+1$ moves to preference j, the candidate with preference $j+2$ moves to preference $j+1$, and so on.

When the main loop of the standard STV algorithm as shown in Figure 4.7 terminates, either (a) all seats are filled, or (b) the number `cc` of remaining candidates is equal to the number of open seats. In case (b), a further step is needed to distribute some or all of the remaining candidates to the equal number of remaining seats. The default is to fill all the remaining seats with the remaining candidates (line 37). Alternatively, one may continue the main STV loop to see if the further candidates get elected (which may leave seats open).

Example 4.5.1. We consider the same situation as in Example 4.3.1, i.e., there are four candidates A, B, C, D for two vacant seats, and the votes to be counted are $[A, B, D], [A, B, D], [A, B, D], [D, C], [C, D]$. The Droop quota in this case is $\mathtt{Q} = \lfloor 5/(2 + 1) \rfloor + 1 = 2$.

In the first iteration of the main loop, candidate A meets the quota and is hence elected. Two of the votes $[A, B, D]$ are erased, the third is a surplus vote. It is transformed into $[B, D]$ by deleting A from the ballots.

In the second iteration no candidate reaches the quota, thus the weakest of the remaining candidates B, C, D is eliminated – which one depends on the kind of tie-breaker used as all three have exactly one first-preference vote at that point. (1) If the tie-break eliminates B, the aforementioned transformed vote $[B, D]$ will be transformed again and will become a vote for D, so that D will be elected in the next iteration. (2) If the tie-break eliminates C, the vote $[C, D]$ will be transformed into a vote for D, and thus D will be elected. (3) If the tie-break eliminates D, then C will be elected, analogously, in the next iteration. In summary, the algorithm reports either $[A, D]$ or $[A, C]$ as the election result but not, for example, $[A, B]$ or $[B, D]$. If the number of second-preference votes is used as a tie-breaker, then B is eliminated first (case 1 above).

The standard STV algorithm has three choice points that produce non-determinism. Different variants of STV resolve them in different way:

1. Who is eliminated if several candidates have the same minimal number of first preferences (line 18)?

2. Who is elected if several candidates have reached the quota (line 23)?

3. How are the votes chosen that are deleted when an elected candidate has more than quote votes (line 27)?

Choice points (1) and (2) are typically handled – to some extent at least – by defining various kinds of tie-break rules. They can also be handled by declaring all weakest candidates eliminated resp. declaring all strongest candidates elected. That, however, is not always possible (there may not be enough open seats). And it can affect the election result in unexpected ways.

Choice point (3) can be eliminated using the notion of fractional votes. Instead of erasing a fraction of the votes that needs to be chosen, the same fraction of each vote is erased and the remaining fraction remains in the ballot box. This is done in many versions of STV used in real-world elections.

The above considerations illustrate that the STV algorithm as presented in this section is not only one but an entire family of vote counting algorithms. There are a number of parameters to play with: the quota, the choice of tie-breakers, placement of candidates once there are as many free seats as remaining candidates.

There are further options that – we argue in Section 4.5.2 – lead to election systems that can no longer be considered part of the STV family.

4.5.2 The CADE-STV Election Scheme

The bylaws of the Conference on Automated Deduction (CADE) specify an algorithm for counting the ballots cast for the election of members to its Board of Trustees [77]. The intention of the bylaws is to design a voting algorithm that takes the voters' preferences into account. The algorithm has been implemented in Java and used by several CADE Presidents and Secretaries in elections for the CADE Board of Trustees. It has also been used by TABLEAUX Steering Committee Presidents, including one of the authors, for the election of members to the TABLEAUX Steering Committee.

Pseudo-code for the CADE-STV scheme is included in the CADE bylaws [77], making it an interesting target for formal analysis. CADE-STV differs from the standard version of STV (shown in Figure 4.7) in several ways:

Quota Instead of the Droop quota, CADE-STV uses a quota of 50% of the votes – independently of the number of seats to be filled. That is, line 5 in Fig. 4.7 is changed to "`Q :=` $round$(`V`/2)".

Empty seats CADE-STV does not fill seats that remain open at the end of the main loop, i.e., lines 36–37 are removed, and "cc > S - e + 1" in lines 10 is changed to "cc > 0".

Restart Each time a candidates c reaches the quota `Q` of first-preference votes and gets elected, the election for the next seat restarts with the original ballot box – with the only exception that the elected candidate c is deleted. Thus, (a) the `Q` votes used to elect c are not erased but are only changed by deleting c, and (b) weak candidates that have been eliminated are "resurrected" and take part in the election again.

That is, (a) the code for erasing votes (lines 26–29) is removed and (b) replaced by code for resurrecting the eliminated candidates.

4.5.3 Applying Bounded Model Checking to CADE-STV

As already explained in Section 4.4.2, we have applied SMT-based bounded model checking to CADE-STV. There, we drew conclusion regarding the applicability of bounded model checking and the use of Z3. In this section, we report what can be concluded from the experiments w.r.t. properties of CADE-STV.

First, the experiments validated that CADE-STV (like standard STV) satisfies Criterion 2 (Def. 4.3.2). But since an exhaustive model search was only done up to a small bound on the number of votes and number of candidates, this does not constitute a full proof.

Second, running our bounded model checker on CADE-STV confirms that, in difference to standard STV, CADE-STV does not satisfy Criterion 1 (Def. 4.3.1), which is closely related to proportional representation.

In addition, we have used SMT-based bounded model checking to check the correctness ratio of CADE-STV voting instances. In this experiment, we generated all possible ballot boxes for CADE-STV up to a certain size and used Z3 to check if the CADE-STV counting results meet Criterion 1 for that instance.

As an example, the number of candidates is fixed to 4, the number of vacant seats is varied from 1 to 3 and the number of voters is set from 3 to 5. The result of the experiment is shown in Tab. 4.1. All possible ballot instances for 4 voters and 2 vacant seats are tested, while for others only the first 100,000 ballot instances are checked. And when CADE-STV fails to seat all the vacant seats, the voting instance is not counted in the correct ratio.

For the one-vacant-seat case, the criterion is met by CADE-STV constantly, because there is no difference between standard STV and CADE-STV (unless no candidate reaches the quota, in which case standard STV picks a random candidate while CADE-STV leaves the seat empty). When there are more than two vacant seats, the criterion is unsatisfiable with CADE-STV because there not enough ballots to support all elected candidates as the quota is 50%. For the two candidates case, CADE-STV sometimes does not reuse the ballot when picking the second candidate, and in these cases the criterion can be satisfied.

Table 4.1: Test First STV Criterion on CADE-STV

Voters \Vacant Seats	1	2	3
3	100%	0%	0%
4	100%	87.2%	0%
5	100%	0%	0%

4.5.4 Effects of the Differences between CADE-STV and Standard STV

Effects of Restart

To illustrate the effect of the restart mechanism in CADE-STV on the election result, we consider an example:

Example 4.5.2. Let us run CADE-STV on Example 4.3.1. First, we compute the majority quota $\mathtt{Q} = 3$. In the first iteration, A has three first preferences, so that A is the majority winner and is seated. Since CADE-STV uses restart, A's votes are not deleted but are redistributed at the end of the first iteration. Now the ballot box contains $[B, D], [B, D], [B, D], [D, C], [C, D]$. Following the algorithm, we observe that now B is the majority candidate with 3 first preference votes and is seated. The election is over, and the election result is $[A, B]$ (which is different from the possible results $[A, D]$ or $[A, C]$ of standard STV).

Indeed, our bounded model checker finds smaller counter examples than the one shown in Example 4.5.2, but these are not as illustrative.

The effect of the differences between standard STV and CADE-STV is further clarified by the following theorem and its corollary: in certain cases, there is no proportional representation in the election results computed by CADE-STV. See also Example 4.5.3 below.

Theorem 4.5.1
If a majority of voters vote in exactly the same way with ballot $[c_1, \dots, c_k]$*, then CADE-STV will elect the candidates* $c_1, \dots, c_k$ *preferred by that majority in order of the majority's preference.*

Proof. Since a majority of voters choose c_1 as their first preference, no other candidate can meet the "majority quota". Thus c_1 is elected in the first round. When redistributing the ballots, each of the majority of ballots with c_1 as first preference have c_2 as second preference. All become first preferences for c_2. Thus candidate c_2 is guaranteed to have a majority of first preferences and is elected in round two, and so on until all vacancies are filled. □

Corollary 1
If the electorate consists of two diametrically opposed camps that vote for their candidates only, in some fixed order, then the camp with a majority will always get their candidates elected and the camp with a minority will never get their candidate elected.

Standard STV does not use the restart mechanism and so it will elect the first ranked candidate of the majority, but will then reuse only the surplus votes and not all votes as done by CADE-STV. Thus the second preference from the majority is not necessarily the second person elected. Consequently, majorities do not rule outright in standard STV.

Effects of High Quota and No Filling of Empty Seats

No matter how many candidates there are and how many seats need to be filled, a candidate can only be seated by CADE-STV if he or she accumulates more than 50% of the votes. Any candidate with less than 50% of the vote is defeated. Thus, CADE-STV obviously violates the fill-all-seats criterion. But because of the high quota it also prevents proportional representation as candidates supported by a large minority can neither be elected via reaching the quota nor via filling seats left empty at the end of the main loop.

In fact, if the high quota of 50% and no filling of empty seats were the only changes w.r.t. standard STV, only a single candidate could be elected because more than 50% of the votes would be used up by electing that candidate. CADE-STV requires the restart mechanism to elect further candidates.

Example 4.5.3. Assume there are 100 seats and two parties nominating candidates $A_1, \ldots, A_{100}$ and $B_1, \ldots, B_{100}$, respectively. Further assume that there are 51% of A-voters and 49% of B-voters. All A-voters vote $[A_1, \ldots, A_{100}]$ and B-voters vote $[B_1, \ldots, B_{100}]$. Standard STV elects $A_1, \ldots, A_{51}, B_1, \ldots, B_{49}$, i.e., the result is a perfect proportional representation.

With a quota of 50% and no filling of empty seats, only A_1 gets elected and then nothing further happens, which is clearly undesirable. But CADE-STV also uses the restart mechanism, therefore, like standard STV, it fills all seats. The result is different, however, because the votes used to elect $A_1, \ldots, A_{51}$ do not get erased. CADE-STV produces the election result $[A_1, ..., A_{100}]$.

The above example again shows that the majority can rule with CADE-STV and there is no proportional representation in that case (Corollary 1).

4.5.5 Observations on the History of CADE-STV

We discuss the history of the CADE-STV scheme because it illustrates the problem of evolving an election scheme without using formally specified semantic criteria and a formal definition of the input to the scheme. It is publicly known that there were lots of discussions among the CADE Trustees over a long period of evolving CADE-STV. But we do not know what the non-public deliberations actually where. The following is based on our interpretation of the publicly available material.

The Violation of Proportional Representation

The CADE-STV voting scheme is the result of a long discussion among the board of trustees that took place in the years 1994–1996. David A. Plaisted published various concerns about the existing voting scheme which can be found on his homepage [99].

One of Plaisted's concerns was that a minority supporting candidates standing for re-election could re-elect these candidates against the wishes of the majority as that majority is not sufficiently coordinated in its behaviour to elect alternative candidates [99]:

> *Of course, one of the main purposes of a democratic scheme is to permit the membership to vote a change in the leadership if there is a need for this. However, the new bylaws make this more difficult in several ways. The problem is that those who are unsatisfied with the scheme will tend to split their votes among many candidates (unless they are so disgusted as to put the trustee candidates at the very bottom of the list), but those who are satisfied will tend to vote for the trustee nominees. This means that the trustee nominees tend to be elected even if only a minority is happy with the scheme.*

We believe that because of Plaisted's concerns the board introduced the high 50% quota and did not include a mechanism for filling seats that remain empty. On first sight, this seems good because it solves the problem illustrated in Plaisted's scenario. But, as explained above, this deviation from the standard STV setup not only violates the fill-all-seats criterion but also the goal of proportional representation (see Example 4.5.3). Thus, the CADE-STV scheme protects the majority at the expense of the minority.

Also, as explained above, if the high quota and the remaining empty seats were the only changes, only a single candidate could be elected. So, in effect, one was forced to change the algorithm further. The result was that the restart mechanism was added to the algorithm, that reuses the original ballot box for each seat and does not erase votes (because then more candidates can be elected, see Example 4.5.3).

There would have been a different solution than using a restart that would have solved Plaisted's problem without restricting proportional representation as much: One could have used Standard STV with an additional rule that – before the main algorithm is started – anybody who does not appear (with arbitrary preference) on at least 50% of the votes is immediately eliminated.

Example 4.5.4. Using the same input ballots as in Example 4.5.3, the algorithm would then elect $[A_1, ..., A_{51}]$, which still suppresses the B minority, but at least gives the A party only those seats that are proportional to the A votes.

Well-formedness and Interpretation of Input

Apparently, during some CADE elections, there was some confusion about the meaning of not listing a candidate at all on a ballot and how that should be translated into input for the CADE-STV voting scheme.

The instruction was given to the voters that not listing a candidate is the same as giving that candidate the lowest possible preference. But that is not the correct interpretation. It is easy to see that for both standard STV and CADE-STV, there is a difference between giving a candidate the lowest possible preference and not listing the candidate at all. For example, if there are candidates A, B, C, then $[A, B]$ is different from $[A, B, C]$. When candidates A and B get eliminated, $[A, B, C]$ turns into a vote for C and may help to elect C, which $[A, B]$ does not. One could transform a ballot of the form $[A, B]$ into an input vote $[A, B, C]$ (and, thus, make them equal by definition). But that only works if a single candidate is missing from the ballot. If more are missing, they would have to be put in the same spot on the ballot, which is not possible. Indeed, CADE-STV does not work correctly if input votes contain candidates with equal preference, i.e., if the pre-condition that a vote is a partial linear order is violated. As that pre-condition was never clearly specified, fixing the problem in CADE-STV was a lengthy process that took severalsjtu

years.

This shows that not formalising the pre-conditions which the input must satisfy is problematic. Besides the possibility of errors or unintended behaviour of the algorithm, it is important that the voters understand how their ballot is transformed into input for the algorithm.

4.6 Conclusion

We have discussed semantic criteria to formalize desired properties of voting schemes. Formal specification and verification do not provide a mechanism for deciding which properties are desirable for a particular vote. But they are methods for the analysis and development of voting schemes.

Our case study demonstrates the importance of formal criteria both for analysis of voting schemes and their evolution and the development process. Semantic criteria need to be explicitly stated. A discussion of voting schemes using anecdotal descriptions of individual voting scenarios is not a good basis for making electoral laws.

Furthermore, we demonstrated that SMT solvers are well-suited to discover bugs in voting schemes. In particular, we have shown that the formalisation of semantic criteria in first-order logic over the theories of integers and arrays is a good choice for SMT-based analysis.

An addition example of applying SMT solvers to the Norwegian and Australian examples (discussed in Chapter 3) is given in Appendix A.1

Chapter 5

Measuring Voter Lines[1]

5.1 Introduction

The question of how to improve the voters' experience when going to the polls is of central importance to many electoral management bodies (EMBs). Good voter experience is often associated with an efficient and professionally organized electoral process, which in turn is expected to lead to elevated levels of voter participation, voter satisfaction, and also trust in the overall election and its outcome. In a recently published report by the US Presidential Commission on Election Administration on the American Voting Experience [16], for example, one of the recommendations calls for *state-of-the-art techniques to assure efficient management of polling places, including tools the Commission is publicizing and recommending for the efficient allocation of polling place resources.* Such state-of-the-art techniques can be roughly divided into two categories: techniques that render polling places more efficient and techniques that measure polling place efficiency. Denmark, for example, uses digital voter registration systems to increase efficiency: By presenting a voter registration with a machine readable barcode, voters can be quickly and efficiently checked of the electoral roll.

This chapter presents a state-of-the-art technique for analyzing the efficiency of a polling place by measuring the times voters are present in and around a polling place. This analysis technique is called the *white boxes*- or simply *wb*-technique. The results of the *wb*-technique are datasets from which we can infer information, for example, when a voter arrived at the polling place, or when he or she left. We conducted the experiments in three

[1]Based on "Measuring Voter Lines" [104], joint work with Carsten Schürmann, accepted and to appear in CeDEM16

elections in Denmark: Danish local elections in 2013, European Parliament election in 2014 and Danish general (parliamentary) election in 2015.

Using statistics, we can deduce valuable information that can assist administrators in rendering polling places more efficient. Examples of the effects of such a method include: Reallocating resources from one polling station to another in the case of a demographic change; purchasing new equipment, such as new ballot boxes, additional registration desks, voting machines or curtains, with the goal to shorten long waiting times; it may also lead to restructuring the layout of polling places to improve flow; or collecting statistical information to quantify voting culture, for example, how many voters skip out of a voting line and leave, how many return again, and how many remain in the polling place after they cast the ballot to wait for friends or chat with acquaintances.

With every redistribution of resources, there is some risk that the quality of the election decreases instead of increases. For example, there are reports from US elections, for example, that after gerrymandering some polling places experienced extremely long waiting times, for example during the 2012 presidential election in Richland County [29]. The data that we collect at polling places can be used to understand these problems better and evaluate the effectiveness of counter measures. It can be used to justify expenses towards more efficient polling place administration and to disarm arguments based on circumstantial evidence, for example unsubstantiated complaints about excessive waiting times.

In the US, there is a critical awareness that such data is invaluable. Per recommendation of the CalTech/MIT voting project [109], several precincts have already collected or are planning to collect queuing data the old fashioned way, i.e. by having election officials count the number of people standing in line at regular intervals, or by handing out pieces of paper to voters that are stamped with arrival and eventually also the departure times; and by tracking individuals throughout the voting process, marking the arrival times at individual service points. In this chapter we show that our *wb*-technique, is on all accounts superior to the manual counting: It is more reliable in that it is virtually free of human error; it is more consistent as information is continuously recorded; it permits (at least in theory) the reconstruction of the path of individual voters through the polling station by trilateralization and it is unobtrusive, because sensors can be installed in the polling place out of sight of the polling officials and voters. Furthermore, in this chapter we show, that the *wb*-technique and the manually collected data can be correlated. In this chapter we show, that *it is possible* *to replace the manual collection of queuing data by a technological solution, while improving the accuracy of the measurement.*

Legal Concerns

According to Danish data protection agency Datatilsynet, media access control (MAC) addresses are considered sensitive (personenfølsom) information, which means that they are protected by national data protection laws. It is illegal to record this information without prior permission of the voter, unless permitted by Datatilsynet. Being aware of the privacy implications for the voter, the DemTech project has applied and was granted permission by the Danish data protection agency to record this data for scientific purposes based on the Danish Data Protection Law (Personendatalov [2]), Chapter 4, Paragraph 6, Section 5.

This chapter is organized as follows. In Section 5.2 we describe the white box-technique. The hardware and software that we chose to implement the technique are described in Section 5.3. Security and privacy considerations are discussed in Sections 5.4 and 5.5, respectively. We describe a pilot study with this technology that we conducted during the 2015 Danish parliamentary election in Section 5.6. We deployed the technology in five polling stations including three in Copenhagen and two in Aarhus. For one polling place in Copenhagen, Holbergskolen, we conducted CalTech/MIT style manual collection of inflow data, which we use to evaluate the quality of our method. Finally, we assess results and describe our preliminary findings in Section 5.7. We conclude that the white box method provides a precise and accurate information to measure waiting times in polling places.

5.2 *White Boxes*-technique

The basic idea behind this technique, is that mobile phones send out wireless packets that can be recorded by a sensor for future analysis. Abstractly, we can describe the data collected this way as a set of observations O, that records wireless packets in the form of pairs

$$(p, t) \in O$$

where p denotes the identifier of a mobile phone and t the time at which a wireless packet was observed. The sensors that record the packets sent

[2] `https://www.retsinformation.dk/forms/r0710.aspx?id=828`

out by mobile phones have only limited range. Therefore, it will become necessary to deploy multiple sensors for one polling station, which entails, that we will also have to combine multiple sets of observations $O_1, \ldots, O_n$ for our analysis. To compute for how long a mobile phone, aka voter was present at a polling place, we have to combine the sets of observations and compute the following presence relation:

$$(p, s, e) \in \text{Presence}(O)$$

where s (e) refers to the earliest (latest) time when (p, s) ((p, e)) was recorded in O. In other words, we can compute precisely for how long each device stayed in a polling place — by subtracting s from e.

Noise: In our pilot study that we describe below in Section 5.6, we have observed that empirically collected data sets contain noise. This noise may be due to other devices emitting wireless packets, such as, for example, routers that are installed in the building hosting the polling station, mobile phones of voting officials or people passing by without voting, or mobile phones of voters that run out of batteries. We remove the noise from the dataset using common statistical methods, in particular, one standard deviation, which is defined as follows.

$$\sigma = \sqrt{E[(T - \mu)^2]}$$

where $\mu = E[T]$ computes the expected value. Data outside the range of $[\mu - \sigma, \mu + \sigma]$ is treated as noise.

Accuracy: Each sensor, will accurately record any wireless packet observed within its range, assuming that the local clock of the sensor is configured correctly. This has two consequences. First, if the waiting queue extends outside the range of a sensors installed, it will not “see” the end of a line, and conversely, a mobile phone may be detected while the voter is still in progress of enqueuing. In the former case, it is important that the voting officials can predict where the line will form and then use sufficiently many sensors to monitor the queue.

Completeness: We note, that not every voter will carry a mobile phone that emits packets. In our experience, this is not problematic, because of the way lines are formed, as long as sufficiently many people in line carry such a phone. Our data shows, that most commonly every third to fourth voter (in extreme cases every eighth voter) carries such a phone. Naturally, the more mobile phones are present in a voting place, the more accurate our estimates for waiting times will become.

Technology	Range
RFID tracking	$< 1m$
Bluetooth tracking	ca. $10m$
WiFi tracking	$< 100m$
UTMS, GSM	$800m - 40km$

Table 5.1: Tracking Technologies

We also note, that the frequency with which mobile phones emit packets depends very much on the phone's operating system. This is because the probe frequency, for example, for WiFi ranges from 4 packets per minute to less than one.

We anticipate a future complication with the *wb*-technique when phones come pre-configured with anonymization enabled. This technique hides the device identifier of the phones during the probing phase. However, this is currently not a concern, as only few phones have this feature enabled [97], in part, because it is so inconvenient to enable it. All notification services, email, messages, etc. must be disabled for this feature to work.

5.3 Implementation

Next, we describe our design choices for implementing the *wb*-technique, in preparation for a real election.

5.3.1 Hardware

The first design choice is which kind of wireless packets we intend to record. There are several options as outlined in Table 5.1. Among these options, we have selected WiFi tracking: Most smartphones and smartwatches have active WiFi on their phones, simply for convenience. This way the phone will try to connect automatically to an access point at home or in the office. When trying to connect, the smartphone periodically emits a probing packet, which we will then record. The range of WiFi phones in the open is around 100m, indoors between 10 and 20m and is therefore better than Bluetooth, which has a much smaller range. In addition many more people use their smartphones to connect to access points than have Bluetooth enabled. We briefly also considered RFID tracking, but discarded this idea because of low range and security concerns. Recording UMTS or GSM signals is in general illegal.

WiFi enabled devices usually transmit on 11 different channels. When trying to connect to an access point they transmit a broadcast packet on all channels, which means that for our system, it is sufficient to listen only to one channel (channel 1) which uses a frequency of 2412 MHz. Note, that the mobile phone has also Bluetooth enabled, there might be some interference between WiFi and Bluetooth.

The second design choice is the particular technology that we shall use to record wireless packets. There are many possibilities, ranging from small computers (for example, Raspberry Pi) to routers. In preliminary experiments, we used a TP link travel router TP-MR3020.[3] These routers have the advantage that we can install OpenWRT (a Linux derivative) on them. They also come with a USB port, which we use to connect a USB drive to record the data persistently, which we found worked well. We also used USB drives in an earlier pilot Danish Municipal election 2013, without any problems.

An alternative technology are 3G antennas, which we used for the European parliament election 2014, where we encrypted and transmitted the data directly to DemTech's server. This had the advantage that in principle, we could provide online queuing data information. However, the 3G connection to the server was less reliable than expected, so that our data collection was spotty at best.

One drawback of these sensors was that they needed to be connected to an external power source, which means that the location for the sensors in a polling place was largely determined by the location of power outlets.

For the 2015 Danish Parliamentary election we therefore started to look for another technology and we chose the TP link travel router TP-MR13U[4] and 16GB SanDisk Cruzer Fit USB Flash drives. This router comes with an embedded battery large enough to power the sensor for 48 hours straight, ample time to be deployed during an election. Just like with the TP-MR3020, OpenWRT can be installed on this sensor and they can be programmed to start and stop collecting data at precise time points. Practically speaking a polling station can be prepared the day before the election, and the sensors can be collected after the polling station closes.

[3] `http://www.tp-link.com/en/products/details/cat-4691_TL-MR3020.html`
[4] `http://wiki.openwrt.org/toh/tp-link/tl-mr13u`

5.3.2 Software

To prepare the routers, we flush the router's firmware and replace it with OpenWRT [5], an operating system based on the Linux kernel that is optimized for network traffic management. Next, we set the wireless interface of each sensor into passive mode, which means that the sensors can only listen. One can think of the sensors as microphones, recording all wireless traffic around them.

OpenWRT supports many of the common networking tools,in particular, tcpdump[6] a tool that we configured to record wireless packets. We configured the sensors in such a way that they would automatically start recording packets at 08:00 and stop at 21:00 on the election day.

5.4 Security

We review the security of our system from the point of assets and vulnerabilities [15] in the polling places. In our experiments, the physical assets are the routers and their respective USB drives. The physical assets' integrity can be violated due to unexpected damage to the sensors, for example, through water, fire, or theft. As logical assets we refer to the router's software and the data collected on the USB flash drive.

Regarding the protection of the physical assets, there is little we can do beyond physically securing them, because the sensors are not monitored. During the experiments described below, we placed the routers in difficult to access places and out of the voters sight.

We therefore direct the focus of this security analysis on the integrity and security of the logical assets. Here, we even assume that the sensor itself is under the adversary's control, and therefore as the primary security mechanism we protect all data on the USB flash drive, by encrypting the relevant partition on the drive. The key for decrypting the flash drive was not stored on the sensor, but kept in our office.

As a secondary security mechanism, we restrict the attack surface of each sensor: the only way to access it is by attaching network cable and logging in through ssh.

The only personnel authorized to access are the system administrators, who are usually not in field. A system administrator has all keys and

[5]`https://openwrt.org/`

[6]`http://www.tcpdump.org/`

thus has the access to all critical data. We argue that the system is secure keeping in mind that an attacker could easily record the same wireless traffic without having to breach the security measures that we have put into place.

Next, we'll address the protection of the collected data. Each recorded packet contains identifying information about the identity of the mobile phone, the so called MAC address. Even though it is possible to reset the MAC address of any such device, it is uncommon for users to do so. Note, that for the purpose of this work, it is important, that

- MAC addresses are properly anonymized to protect the identity of the voter,
- only data relevant to our experiments is kept, including anonymized MAC addresses, time stamps, and signal noise ratios, and
- the anonymization of MAC addresses is deterministic, because we need to correlate packets across multiple sensors.

For practical purposes, during the voting day, all data is recorded. Any anonymization and analysis of the data is done post-election.

The data collected during this pilot must be classified as "sensitive" because MAC addresses are considered personal data. To do an appropriate security analysis of our design, we consider which possible adversary might be interested in stealing the data we collect. An adversary may want

- to steal data out of curiosity,
- to gain access to recorded information,
- to discredit the election commission,
- or to interrupt the pilot.

It is therefore important to take the necessary technical and operational security precautions. However, in general, the security threat is at most moderate, as many polling places are within the range of wireless routers that could be reprogrammed to collect similar kinds of information. Every smart phone can be programmed to do the same.

We use disk encryption (AES based, SHA1, 256 bit) to deter attackers and to secure the data stored on the USB drive on the sensor. The disk encryption subsystem is part of the Linux kernel (`dm -crypt`). This way, we can guarantee that all data on the USB drive is encrypted and thus rendered useless for an attacker stealing it.

5.5 Privacy

The MAC address for a network interface, is factory preset for each networked device. This includes smartphones, but also standard networking hardware that can be found in laptops and other computers. A MAC address contains 6 octets, as shown in Figure 5.1, where the first 3 octets identify the organization that issued this MAC. Using this and the remaining 3 octets, it is possible to identify the device that owns the MAC.

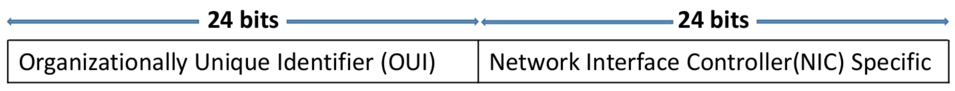

24 bits
24 bits
Organizationally Unique Identifier (OUI)
Network Interface Controller(NIC) Specific

Figure 5.1: MAC address

For most devices, in particular Apple's iPhone or Android phones, the MAC address can be reset by the user to any randomly chosen 6 octets [7]. For example c2:31:9d:d0:30:e8 is a valid MAC address. As each octet is 8 bit, there are 2^{48} unique MAC addresses. Standard anonymization techniques, for example, by computing the SHA256 digest for the MAC address from a voter's phone provide only low levels of security, because a simple dictionary attack would allow an adversary to relate the anonymized identifier to the original MAC address. Therefore, we chose to use HMAC (keyed-hash message authentication code) to hide the MAC addresses before we published the raw data, so that the HMAC key protects against such a dictionary attack.

For reliability, when we use tcpdump to capture the wireless packets, we do not filter out the packet header during recording. Instead, we store the whole packet, and remove the packet's body offline. The secrecy of each packet's content depends on the type of security method used for wireless communication, for example WEP, WPA, or WPA2. In addition, most voters will not be able to connect to any of the local networks in the polling station, which means that all the captured packets will only be handshake beacons, instead of payload carrying network packets, which arise when checking email or surfing the web.

Our security and privacy measures are not designed to protect against inside attacks, which means if the keys are misused by an inside attacker, the attacker may gain free access to the data on the USB drive and recover

[7]Some special MACs are reserved

the original MAC addresses from the anonymized ones. If the attacker is in possession of the secret key used to compute the HMAC, he will be able to brute force the original MAC address.

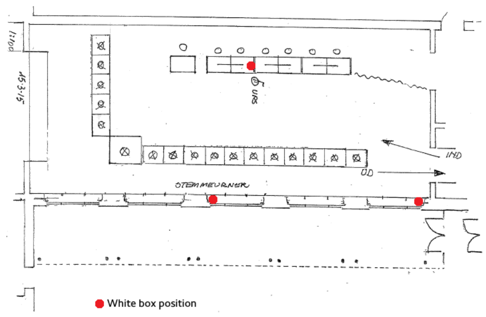
5.6 Field Study

In this section we describe the 2015 pilot project, where we recorded network traffic using the white boxes method and recorded them manually following the CalTech/MIT style.

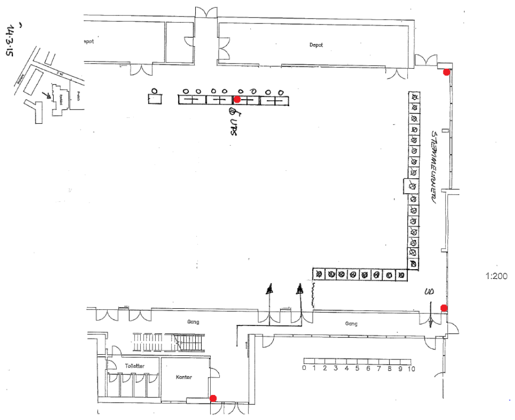
The general parliament election was held in Denmark on 18 June 2015 from 09:00 to 20:00 to elect the 179 members of the Folketing (Danish parliament). We installed our white box technology in 5 polling places in two cities, Aarhus and Copenhagen, and deployed in total 18 white boxes with a minimum of 3 white boxes installed in each polling station. Out of the 5 polling places, 3 were located in Copenhagen, namely Holbergskolen, Bellahøj Skole and Islands Brygge Skolen, and 2 were located in Aarhus, namely Møllevangskolen and Frederiksbjerg Hallerne.

5.6.1 Installation

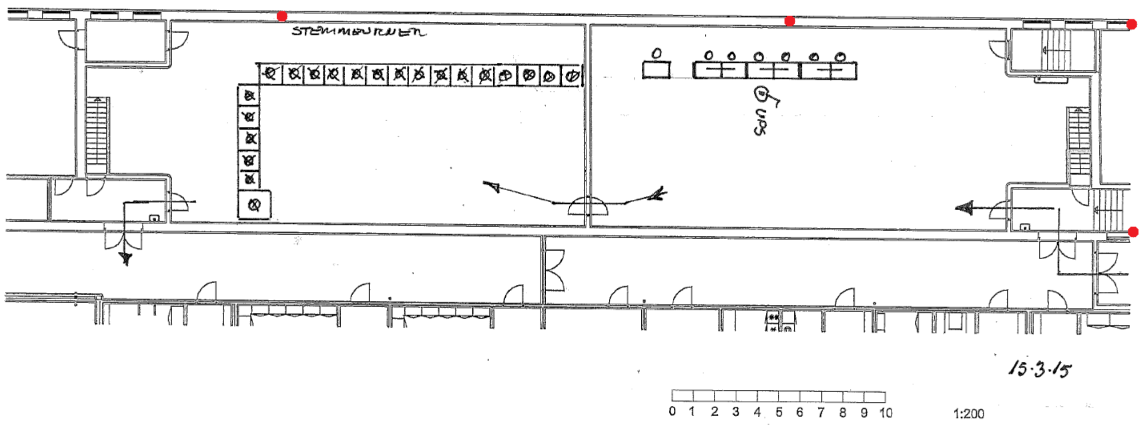
The position for each white box in a polling place was carefully chosen, in an effort to ensure that the combination of white boxes cover the entire polling station and the area where there queues were expected to form. In some polling places we placed white boxes in such a way to achieve a higher level of redundancy, useful for the case sensor failure. As shown in Figure 5.2a, 3 sensors were placed in Holbergskolen polling place in Copenhagen, where the bottom two sensors were placed to record the mobile devices before voters go through the entrance (IND) and after they leave (UD). The small circles mark the registration desks, the first service point. After registration, the voters received a blank ballot and proceeded to the second service point, waiting for an empty voting booth, marked in Figure 5.2a by a crossed out circle. The queues formed alongside the building, along the dotted line in the picture.



(a) Holbergskolen



(b) Islands Brygge Skolen



(c) Bellahøj Skole

Figure 5.2: Layout Maps of White Boxes in Copenhagen

On the day before the election, we deployed five teams who visited one of the five polling places each, and installed 3-4 sensors. The sensors were programmed to record from 08:00 to 21:00. The sensors were completely autonomous and did not require any servicing during election day. Each team collected the sensors after close of polling station and returned them to the lab for further processing.

5.6.2 Data Preprocessing

The data collected by each sensor consists of a list of packets. These lists contain several millions of packets each. Each packet is described by a time stamp, an anonymized MAC address, and a value representing signal strength.

Figure 5.3 depicts a plot of one of these sensors, before noise was removed. The x axis describes time and the y axis the identifier of the mobile phone, sorted in order of first appearance. As we can see, several sensors leave long lines to the right; this may be due to routers in proximity to the polling station being turned on, or other reasons.

First, we tweak the data a little:

1. We preprocess the data in the following format, summarizing all observations at times $t_1...t_n$ of the same identifier into one tuple $(p, \{t_1, \ldots, t_n\}, l)$. l refers to the duration an identifier was within scope of the polling station $l = t_n - t_1$.

2. We remove all tuples from this set, where $l = 0$.

3. We remove all tuples, where $60min < l$.

4. We remove all tuples, recorded $30min$ before the polling station opens, i.e. where $t_1 <$ `08:30`.

5. We remove all tuples, recorded $30min$ after the polling station closes, i.e. where `20:30` $< t_n$.

6. We remove all tuples corresponding to *spurious* identifiers, where $l < 1min$.

7. We remove all tuples corresponding to *transient* identifiers, i.e. there exists an i, such that $10min < t_{i+1} - t_i$.

8. Finally, we apply the noise removal techniques described earlier, which significantly improved the overall quality of the data. With a running average (deviation) μ (σ) of the duration l for 10 minute intervals (empirically determined), we remove in addition all those tuples for which $l < \mu - \sigma$ or $\mu + \sigma < l$.

It is also conceivable to preprocess the data set even further: For example, for each wireless packet, we could determine the manufacturer information from the MAC address (before anonymization). Based on this information, we could decide to keep or remove the tuple. This is possible, because the first 24 bits of a MAC address identify the manufacturer as discussed in Section 5.5 (unless the MAC address was reset). There are online inverse MAC-address lookup services.

When we black-listed the five common router manufacturers, and removed all packets from those devices from our data set, we observed, that our method produced nearly identical results. This means, that our data processing step effectively removed all black-listed devices automatically.

We also observed, that our sensors sometimes behave erratically. The sensor No.7's unprocessed data depicted in Figure 5.3, for example, appeared to have malfunctioned as it stopped recording packets for about one hour. The damage could be mitigated, because the other sensors in polling places worked well. We remark, that this was also the only malfunctioning that we observed.

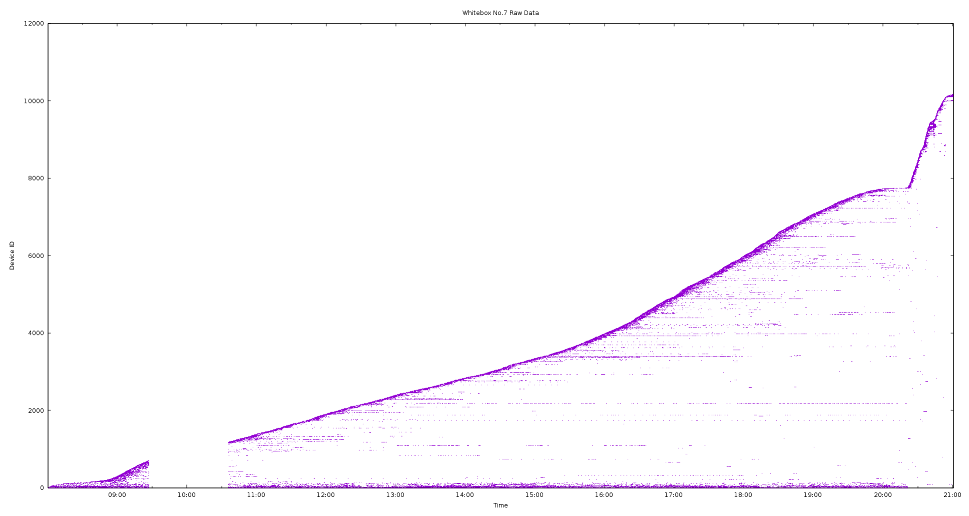


Figure 5.3: White Box No.7 Data

5.6.3 Devices

In Denmark, it is common practice to publish statistical information about each election on the internet. The official election homepage [48], for example, lists how many voters voted in each polling place. We use this information to approximate, how many voters carried a WiFi enabled mobile phone (see Table 5.2). The overall penetration of smart phones in Denmark was 59% in 2013 [8].

Polling Places	Voters	Devices	Percentage
Holbergskolen	4997	1333	26.68%
Bellahøj Skole	5789	1942	33.55%
Islands Brygge Skolen	7931	3494	44.05%
Møllevangskolen	8043	3132	38.94%
Frederiksbjerg Hallerne	10578	3798	35.90%

Table 5.2: Device Capture Ratio

5.6.4 Queues

We now discuss how to interpret the data that we have collected using the polling station at Islands Brygge Skolen as an example. The results for the other polling places are given in Appendix A.2, Appendix A.3 and Appendix A.4.

First, we visualize the data we have collected. As previously stated, our data set consists of a set of tuples of the form (p, t_1, t_n, l), where p stands for the anonymized identifier of each mobile phone, t_1 the point in time when the device was detected first, and t_n when it was seen last. Below we refer to this data set as $\mathcal{D}$. l refers to the total voter's waiting time (time spend in the polling place). The graph in Figure 5.4 visualizes this — the vertical axis ranges over device IDs ordered by t_1, and the horizontal axis ranges over time. For each device, we mark t_1 by a blue and t_n by a red dot.

Next, we describe two analysis techniques, for computing the average time that voters spent in a polling place. First, we use the method of averaging, which is described in Section 5.6.4, and second, we use a method based on Little's theorem [89] in Section 5.6.4.

Method based on averaging

With this method, we compute the running averages of the time spent in a polling place, using a sliding window of 10 minutes, 30 minutes, and 1 hour, respectively:

[8]https://en.wikipedia.org/wiki/List_of_countries_by_smartphone_penetration

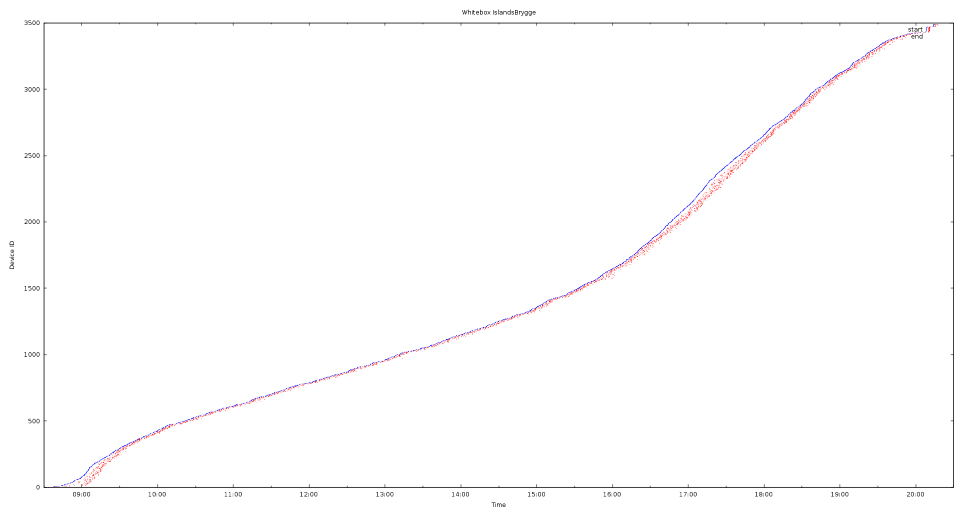


Figure 5.4: Device Appearance in Islands Brygge Skolen Copenhagen

$$\mu_{len}(t) = \frac{\sum_{l \in L} l}{|L|}$$

where L is defined as $\{l | (p, t_1, t_n, l) \in \mathcal{D}$ and $t - \frac{len}{2} \leq t_1 \leq t + \frac{len}{2}\}$ and len is the size of the window. Figure 5.5 depicts a green, blue, and red graph for the polling place at Islands Brygge Skolen representing the running averages for a 10 minute, 30 minute, and one hour window, respectively. The horizontal axis denotes time and ranges from 08:30 to 20:30. The vertical axis denotes the average waiting time, in seconds.

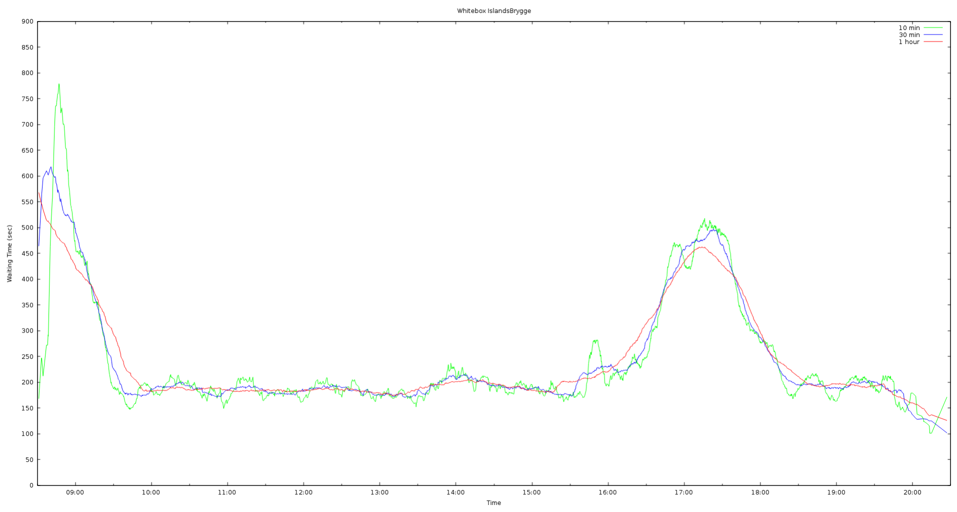


Figure 5.5: Average Time (Averaging Method) in Islands Brygge Skolen Copenhagen

Method based on Little's theorem

In queuing theory, Little's theorem is widely applied. Little's theorem states [89]: *"The average number of customers in a system (over some interval) is equal to their average arrival rate, multiplied by their average time in the system."* Based on this theorem, we can compute the average time the voters spent in the polling station as

$$\mu_{len}(t) = \frac{V_{len}(t)}{\lambda_{len}(t)},$$

where *len* stands for the size of the window, as above. V_{len} is the average number of voters in the polling station at time t, and λ_{len} is the arrival rate also at time t.

The accumulated inflow values, outflow values, and their difference, which corresponds to the number of voters present in the polling station at a particular time, are plotted in Figure 5.6. The grey, green, and blue graphs, correspond to inflow, outflow, and number of voters, respectively.

Figure 5.7 shows the average time voters spent in the Islands Brygge Skolen polling station for any particular point in time during the voting day. The color schemes are consistent with those used above, green refers to a 10 minute window, blue to a 30 minute window, and red to a one hour window.

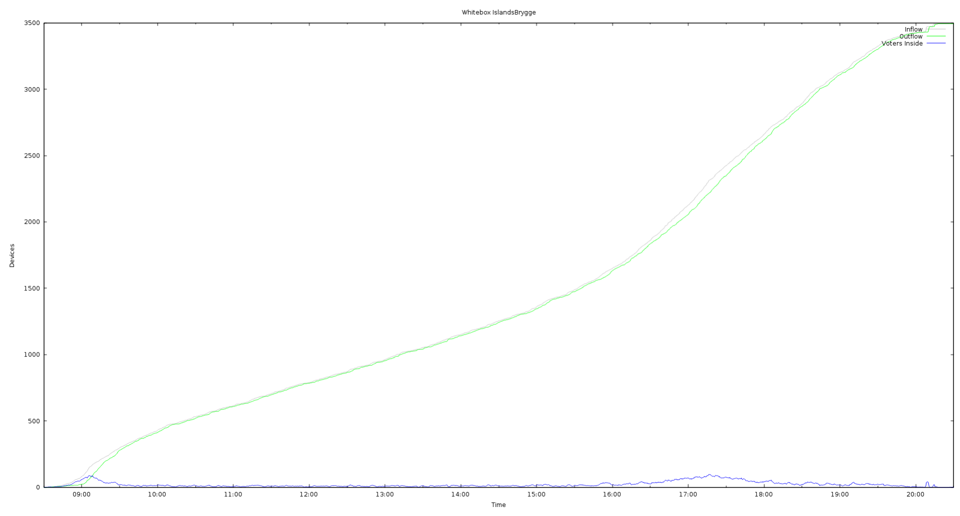


Figure 5.6: Device Flow in Islands Brygge Skolen Copenhagen

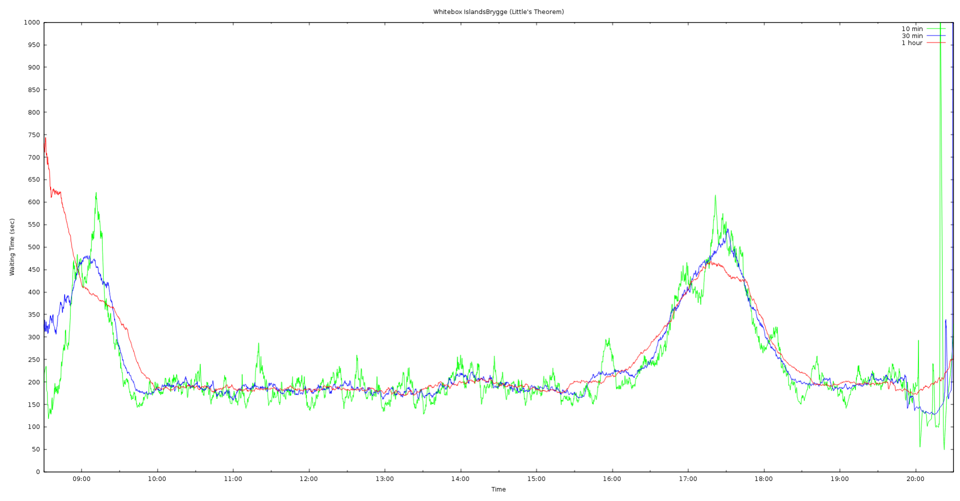


Figure 5.7: Average Time (Little's Method) in Islands Brygge Skolen Copenhagen

5.6.5 Manual Count

In order to validate our method, we conducted in addition to the automatic method a manual count in parallel, following the CalTech/MIT recommendations, for assessing waiting times in polling places. In this section, we report which data we collected, and how to compare the white box method with the manual method. The CalTech/MIT recommendation proposes two different kinds of manual data collection.

1. The first method asks an observer to follow and record the individual activities of random voters throughout the polling station. This includes, arrival times, for example, at the end of the queue, or different service points, and the departure of the voter from the polling station. Table 5.3 depicts a fragment of the log that we manually recorded at the polling station, Holbergskolen.

No.	Description	Check-in	Arrival registration	Leave registration	Enter booth	Leave booth	Ballot cast
16	green coat	10:51:40	10:53:00	10:53:15	10:53:24	10:54:09	10:54:14
17	purple jacket	10:59:45	11:00:35	11:00:51	11:00:57	11:02:00	11:02:11
18	lime jacket	11:10:06	11:10:58	11:11:20	11:11:30	11:12:22	11:12:28
19	red coat	11:19:10	11:19:24	11:19:40	11:19:48	11:20:26	11:20:28
20	white pants	11:28:10	11:28:30	11:28:44	11:28:46	11:29:16	11:29:20

Table 5.3: Example Data of Manual Count in Holbergskolen

2. The second method for data collection requires an observer to count the number of arriving voters in 10 minute intervals. The two leftmost columns of Tables 5.5–5.7 display this information for 4 different polling places. Due to a lack of observers, we were not able to record this data for the polling station at Frederiksbjerg Hallerne.

Next, we discuss the comparison between the white box method and the manual methods.

Queues

Here, we compare the data that we have collected in Holbergskolen using the white box method to Table 5.3 above. From this data, we compute the time that each voter spent in the polling place, by subtracting "check-in time" from the "ballot cast time". Using the averaging methods, we compute the running averages using a window size of 30 minutes and one hour, respectively. We remark that a window size of 10 minutes did not provide usable information, in part because of a lack of observations. These two graphs are depicted in Figure 5.8 and Figure 5.9, using the colors light blue (30 minutes), and gray (1 hour). The remaining three plots (green, blue, and red) display the white box data for this polling place using the averaging method (see Section 5.6.4) and Little's method (see Section 5.6.4).

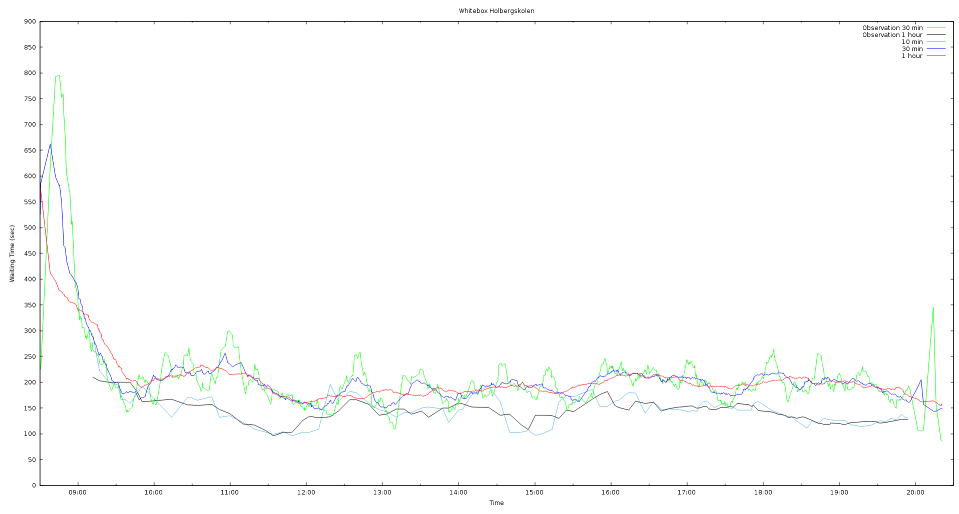


Figure 5.8: Comparison of White Box (Averaging Method) and Manual Count in Holbergskolen Copenhagen

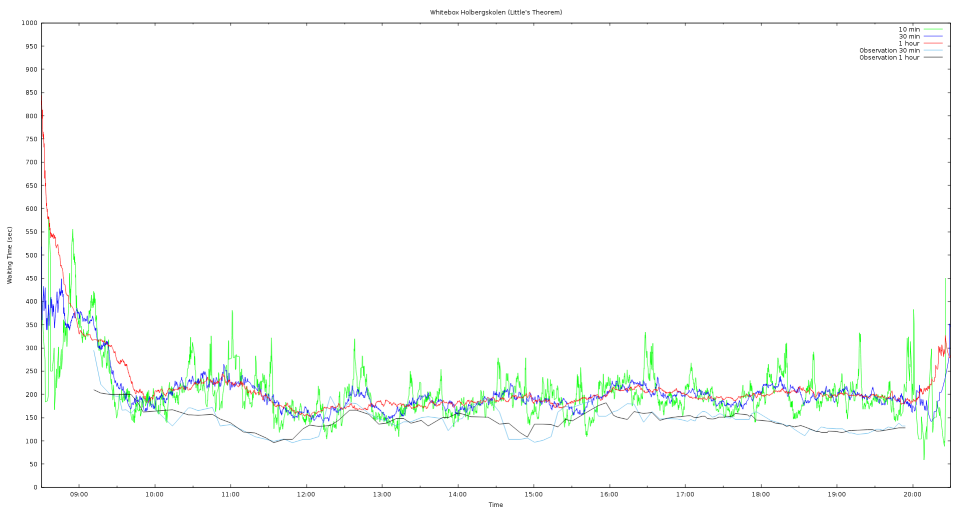


Figure 5.9: Comparison of White Box (Little's Method) and Manual Count in Holbergskolen Copenhagen

Devices

Here, we compare the second type of data that we collected manually with the data recorded using the white boxes. There is only one polling station, namely Holbergskolen, for which we recorded inflow manually during the whole day. Figure 5.10 depicts the inflow and outflow, recorded by our sensors in green and gray, respectively, and the inflow recorded manually in red. Note, that we did not record manually the outflow of voters from a polling station.

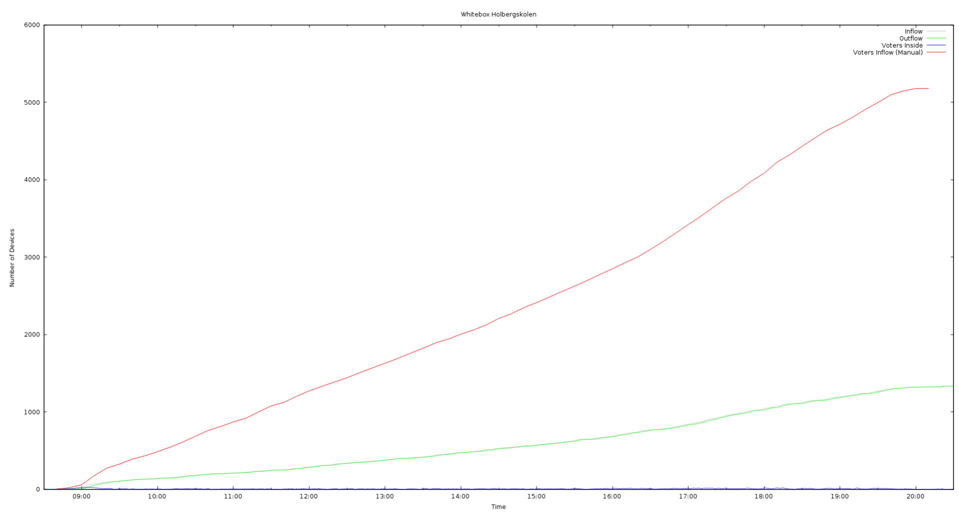


Figure 5.10: Device Flow in Polling Station Holbergskolen Copenhagen including manual count

Figure 5.11 presents the same information but in a slightly different form. The red plot describes the average percentages of voters who carried a mobile phone, in 10 minute intervals.

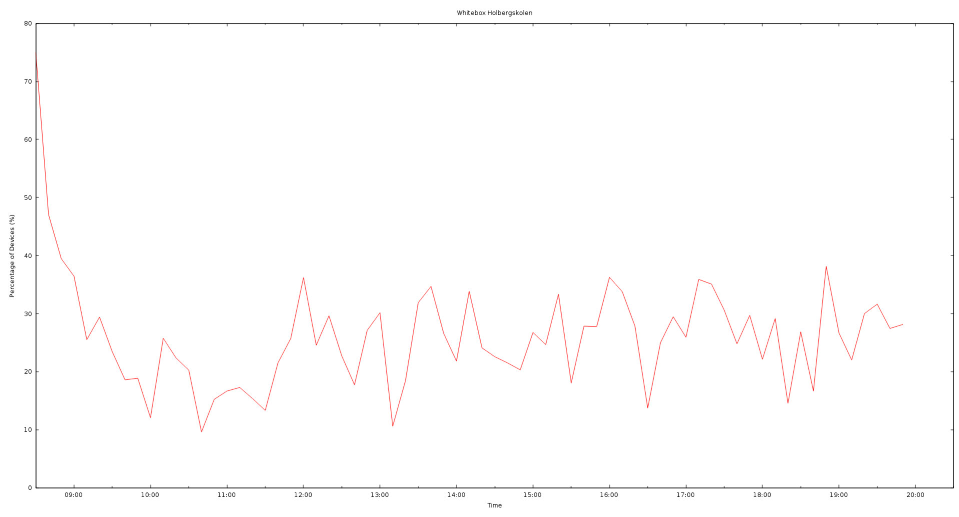


Figure 5.11: Device Capture Ratio from 8:30 to 20:10 in Holbergskolen Copenhagen

In addition to the inflow collected from Holbergskolen (Table 5.4), we also recorded the inflow for one hour period between 16:00 and 17:00 for three further polling places, including Bellahøj Skole (Table 5.5), Islands Brygge Skolen (Figure 5.6), and Møllevangskolen (Table 5.7). This allows us to compare the inflows between all four polling places during this hour.

Time	Voters	Devices	Percentage
...	...	...	...
16:00 - 16:10	80	29	36.25%
16:10 - 16:20	74	25	33.78%
16:20 - 16:30	97	27	27.84%
16:30 - 16:40	102	14	13.73%
16:40 - 16:50	108	27	25.00%
16:50 - 17:00	112	33	29.46%
...	...	...	...

Table 5.4: Device Capture Ratio in Holbergskolen Copenhagen

Time	Voters	Devices	Percentage
16:10 - 16:20	90	28	31.11%
16:20 - 16:30	85	33	38.82%
16:30 - 16:40	132	42	31.82%
16:40 - 16:50	122	43	35.25%
16:50 - 17:00	118	47	39.83%
17:00 - 17:10	130	48	36.92%

Table 5.5: Device Capture Ratio in Bellahøj Skole Copenhagen

Time	Voters	Devices	Percentage
16:00 - 16:10	107	56	52.34%
16:10 - 16:20	183	77	42.08%
16:20 - 16:30	145	79	54.48%
16:30 - 16:40	200	84	42.00%
16:40 - 16:50	211	93	44.08%
16:50 - 17:00	193	88	45.60%

Table 5.6: Device Capture Ratio in Islands Brygge Skolen Copenhagen

Time	Voters	Devices	Percentage
16:00 - 16:10	138	49	35.51%
16:10 - 16:20	160	64	40.00%
16:20 - 16:30	170	76	44.71%
16:30 - 16:40	173	76	43.93%
16:40 - 16:50	165	73	44.24%
16:50 - 17:00	201	83	41.29%

Table 5.7: Device Capture Ratio in Møllevangskolen Aarhus

5.7 Findings

This section describes some findings from the white box data.

1. The white box method provides a precise and accurate method to collect information for measuring waiting times in polling places. We have observed and recorded data in one polling station in two different ways, using our white box method and a manual counting method. Figure5.9 reports our observations. We can see that both data sets (green and light blue) follow roughly the same trend, but that the automatic method recorded overall higher waiting times than the manual method. We attribute this difference to our setup, where mobile phones are detected earlier by our white box sensors because they record mobile phones while they are approaching the polling station and not just upon arrival. This can be compensated for by using more sensors but with restricted range. To a lesser extent, it may also be due to the inaccuracy inherent in manual data collection and the low sampling rate.

2. We can also refine the setup to collect more detailed data, for example, by increasing the number of sensors, or by using location algorithms, such as trilateralization. It might even be possible to track individual mobile phones or to predict where a line will form. Such extensions are however beyond the scope of this chapter.

3. Some polling places experienced an increase in waiting times between 16:00 and 18:00. For example, at Islands Brygge Skolen waiting time doubled around 17:00. We suspect that this because many voters voted after work. To alleviate these service bottlenecks, our findings indicate that either more registration desks or more curtains are needed.

4. Our findings also show that every polling station has a characteristic service time. This is the minimal amount time a voter requires to traverse the polling station. In the Islands Brygge Skolen polling station, the minimal service time is around 200 seconds, whereas in Bellahøj Skole, it is closer to 270 seconds. Factors that affect the service time include polling station layout and processing speed. Election officials may respond to such observations for future elections and monitor effectiveness.

5. The white box data can also be used to analyze the percentages of the voting populations, who have smartphones and this information can be correlated with voter participation. It is also possible to conduct a further statistical analysis, for example by correlating our data with the official voter registration data.

6. In general, the averaging method and Little's method provide very similar results, although the latter appears to be more fine-grained. In particular, the plots obtained by Little's method in the Figures above and the Appendix below show interesting periodic fluctuations, that we will analyze in future work.

7. Not surprisingly, all polling places show a peak in waiting times at the beginning of the voting day. We observed long queues forming outside polling places, before they opened. After the voting had started, and polling places had been operating at full capacity for a little while, the waiting times dropped dramatically.

8. In Frederiksbjerg Hallerne (as shown in Appendix A.3, Figure A.10), there is a dip in the graph around 10am, which coincides with the

malfunction of sensor No. 7 placed at the entrance at the polling place, as discussed in Figure 5.3. This means that area covered by our sensors was somewhat smaller during this time, resulting in reduced waiting times in this polling place.

Bibliography

[1] Ben Adida. *Advances in cryptographic voting systems.* PhD thesis, Massachusetts Institute of Technology, 2006.

[2] B Alpern and FB Schneider. Defining Liveness. *Information processing letters*, 1985.

[3] Stephen Ansolabehere and Daron Shaw. Assessing (and fixing?) Election Day lines: Evidence from a survey of local election officials. *Electoral Studies*, 41:1–11, 2016.

[4] Tigran Antonyan, Seda Davtyan, Sotiris Kentros, Aggelos Kiayias, Laurent Michel, Nicolas Nicolaou, Alexander Russell, and Alexander A. Shvartsman. Automating Voting Terminal Event Log Analysis. *Electronic Voting Technology Workshop*, 2009.

[5] Hans van Maaren Armin Biere, Marijn Heule and Toby Walsh. Bounded Model Checking. In *Handbook of Satisfiability*, volume 185, pages 457–481. 2009.

[6] Kenneth J. Arrow. A Difficulty in the Concept of Social Welfare, 1950.

[7] a.S. Rao and M.P. Georgeff. Modeling rational agents within a BDI-architecture. *Readings in agents*, pages 317–328, 1997.

[8] Tomáš Babiak, Mojmír Křetínský, Vojtěch Řehák, and Jan Strejček. LTL to Büchi Automata Translation: Fast and More Deterministic. volume 061, pages 95–109. 2012.

[9] Christel Baier and Joost-Pieter Katoen. *Principles Of Model Checking*, volume 950. 2008.

[10] Musard Balliu, Mads Dam, and Gurvan Le Guernic. Epistemic temporal logic for information flow security. *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security - PLAS '11*, pages 1–12, 2011.

[11] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. *ACM SIGSOFT Software Engineering Notes*, 31:82, 2006.

[12] David Basin, Felix Klaedtke, Srdjan Marinovic, and E Zălinescu. Monitoring of temporal first-order properties with aggregations. *Runtime Verification*, 2013.

[13] David Basin, Felix Klaedtke, and Samuel Müller. Monitoring security policies with metric first-order temporal logic. *Proceeding of the 15th ACM symposium on Access control models and technologies - SACMAT '10*, page 23, 2010.

[14] David Basin, Felix Klaedtke, and Samuel Müller. Policy monitoring in first-order temporal logic. *Computer Aided Verification*, 2010.

[15] David Basin, Patrick Schaller, and Michael Schlapfer. *Applied Information Security.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[16] Robert F. Bauer and Benjamin L. Ginsberg. The American Voting Experiance: Report and Recommendations of the Presidential Commission on Election Administration. Technical Report January, 2014.

[17] Patrick Baxter, Anne Edmundson, Keishla Ortiz, Ana Maria Quevedo, Samuel Rodríguez, Cynthia Sturton, and David Wagner. Automated Analysis of Election Audit Logs. *Electronic Voting Technology Workshop*, 2012.

[18] Bernhard Beckert, R Goré, and C Schürmann. Analysing Vote Counting Algorithms via Logic. *Automated Deduction–CADE-24*, pages 135–144, 2013.

[19] Bernhard Beckert, Rajeev Goré, Carsten Schürmann, Thorsten Bormer, and Jian Wang. Verifying Voting Schemes. *Journal of Information Security and Applications*, 19:115–129, 2014.

[20] Francesco Belardinelli and A Lomuscio. First-order linear-time epistemic logic with group knowledge: An axiomatisation of the monodic fragment. *Fundamenta Informaticae*, pages 1–16, 2011.

[21] Pierre Bieber and Frederic Cuppens. Computer Security Policies and Deontic Logic. *Computer Security Policies and Deontic Logic*, 1994.

[22] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using SAT procedures instead of BDDs. *Proceedings of the 36th annual ACM/IEEE Design Automation Conference (DAC99)*, pages 317–320, 1999.

[23] Armin Biere, Alessandro Cimatti, Edmund M Clarke, and Yunshan Zhu. Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume TACAS 99, pages 193–207, London, UK, 1999. Springer-Verlag.

[24] N. Bjorner and L. de Moura. Z3: An efficient SMT solver. *[cited 2010 July]; Available from:http://research.microsoft.com/projects/Z3*, 4963 LNCS:337–340, 2007.

[25] Steven J Brams. Voter Sovereignty and Election Outcomes. (November), 2003.

[26] Felix Brandt, Vincent Conitzer, and Ulle Endriss. Computational Social Choice. In G. Weiss, editor, *Multiagent Systems*. MIT Press, 2012.

[27] DFC Brewer and MJ Nash. The Chinese Wall Security Policy. *Security and Privacy*, pages 206–214, 1989.

[28] Jacob West Brian Chess. *Secure Programming with Static Analysis*, volume 53. Addison-Wesley Professional, Cambridge, 2007.

[29] Duncan a Buell. An Analysis of Long Lines in Richland County, South Carolina. *Presented as part of the 2013 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections*, 1(1):106–118, 2013.

[30] Muhammed Fatih Bulut, Yavuz Selim Yilmaz, and Murat Demirbas. Crowdsourced Line Wait Time Estimation using Smartphones. *Mobicase*, 2012.

[31] Craig Burton, Chris Culnane, James Heather, Thea Peacock, Peter Y. A. Ryan, Steve Schneider, Sriramkrishnan Srinivasan, Vanessa Teague, Roland Wen, and Zhe Xia. Using Prêt à Voter in Victoria State Elections. *Electronic Voting Technology Workshop*, 2012.

[32] BVerfG. Provisions of the Federal Electoral Act from which the effect of negative voting weight emerges unconstitutional, 2008.

[33] David Chaum, Peter Y A Ryan, and Steve Schneider. A practical voter-verifiable election scheme. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3679 LNCS(December):118–139, 2005.

[34] Hao Chen and David Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. *CCS '02 Proceedings of the 9th ACM conference on Computer and communications security*, pages 235–244, 2002.

[35] Jeremy Clark and Urs Hengartner. Selections : Internet Voting with Over-the-Shoulder Coercion-Resistance. *Financial Cryptography and Data Security*, pages 47–61, 2012.

[36] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded Model Checking using SAT Solving. *Journal of Formal Methods in System Design*, 19(1):7–34, 2001.

[37] Edmund M Clarke and E Allen Emerson. Design and Synthesis of synchronization skeletons for branching time temporal logic. *Logics of Programs Workshop, New York, May 1981*, 131:52–71, 1982.

[38] MR Clarkson and Bernd Finkbeiner. Temporal Logics for Hyperproperties. *Principles of Security and Trust*, pages 1–20, 2014.

[39] Dermot Cochran. Secure internet voting in Ireland using the Open Source Kiezen op Afstand (KOA) remote voting system. (March), 2006.

[40] Dermot Cochran and Joseph R Kiniry. Verification of Vote Counting in Irish Elections. pages 1–13.

[41] Robert B. Cooper. *Introduction to Queueing Theory.* Elsevier North Holland, New York, 2nd ed. edition, 1981.

[42] Arel Cordero and David Wagner. Replayable Voting Machine Audit Logs. *Electronic Voting Technology Workshop*, 2008.

[43] Scott A. Crosby and Dan S. Wallach. Efficient Data Structures for Tamper-Evident Logging. *Proceedings of the 18th USENIX Security Symposium*, 2009.

[44] Chris Culnane, James Heather, Steve Schneider, and Zhe Xia. Software Design for VEC vVote System. Technical report, 2013.

[45] Chris Culnane, Peter Y A Ryan, Steve Schneider, and Vanessa Teague. vVote: a Verifiable Voting System. Technical report, 2014.

[46] Chris Culnane and Steve Schneider. A Peered Bulletin Board for Robust Use in Verifiable Voting Systems. *IEEE 27th Computer Security Foundations Symposium (CSF 2014)*, pages 169–183, jan 2014.

[47] Marco Daniele, Fausto Giunchiglia, and Moshe Y Vardi. Improved Automata Generation for Linear Temporal Logic. In *Link.Springer.Com*, volume 1633, pages 249–260. 1999.

[48] Danmarks Statistik. Oversigt over valgkredse i Danmark, 2015.

[49] S Demri and P Gastin. Specification and verification using temporal logics. *Modern applications of automata theory*, pages 1–39, 2011.

[50] J. DeTreville. Binder, a logic-based security language. *Proceedings 2002 IEEE Symposium on Security and Privacy*, pages 105–113, 2002.

[51] E-voting.cc. E-Voting Map, 2015.

[52] Doron Peled Edmund M. Clarke, Orna Grumberg. *Model Checking*. MIT Press Cambridge, MA, USA, 1999.

[53] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. *Theory and Applications of Satisfiability Testing*, pages 502–518, 2004.

[54] B Ferris, K Watkins, and A Borning. OneBusAway: results from providing real-time arrival information for public transit. *Proceedings of the SIGCHI Conference on Human Computer Interaction*, pages 1807–1816, 2010.

[55] Michael Fisher. *An Introduction to Pratical formal Methods Using Temporal Logic*. John Wiley & Sons, Inc., 2011.

[56] Michael Fisher and Michael Wooldridge. Temporal Reasoning in Agent-Based Systems. chapter Fisher2005, pages 469–495. 2005.

[57] Dov Gabbay. The declarative past and imperative future. *Temporal Logic in Specification*, 398:409–448, 1989.

[58] Dov Gabbay, Amir Pnueli, Saharon Shelah, and Jonathan Stavi. On the temporal analysis of fairness. *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '80*, pages 163–173, 1980.

[59] Malay Ganai and Aarti Gupta. Accelerating High-level Bounded Model Checking. In *2006 IEEE/ACM International Conference on Computer Aided Design*, pages 794–801. IEEE, nov 2006.

[60] Deepak Garg. *Proof theory for authorization logic and its application to a practical file system.* PhD thesis, Carnegie Mello University, 2009.

[61] Paul Gastin and Denis Oddoux. Fast LTL to Büchi Automata Translation. *Computer Aided Verification 2001*, 2102(1):53–65, 2001.

[62] Fausto Giunchiglia and Toby Walsh. A theory of abstraction. *Artificial Intelligence*, 57(2-3):323–389, 1992.

[63] Kristian Gjøsteen. Analysis of an internet voting protocol. pages 1–43, 2010.

[64] Kristian Gjøsteen. The Norwegian Internet Voting Protocol. *E-Voting and Identity*, pages 1–13, 2011.

[65] Patrice Godefroid. Model checking for programming languages using VeriSoft. *POPL {'}97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1(January):174–186, 1997.

[66] JA Goguen and J Meseguer. Security policies and security models. *1982 IEEE Symposium on Security and Privacy*, pages 11 – 20, 1982.

[67] Jens Groth. Evaluating security of voting schemes in the universal composability framework. *Applied Cryptography and Network Security*, pages 1–18, 2004.

[68] Sven Heiberg, Peeter Laud, and Jan Willemson. The Application of I-voting for Estonian Parliamentary Elections of 2011. *E-Voting and Identity*, 2011.

[69] Sven Heiberg, Arnis Parsovs, and Jan Willemson B. Log Analysis of Estonian Internet Voting 2013-2014. *VoteID 2015*, 9269:19–34, 2015.

[70] Sven Heiberg, Arnis Parsovs, and Jan Willemson. Log Analysis of Estonian Internet Voting 2013-2014. *E-Voting and Identity, Voteid 2015*, 9269:19–34, 2015.

[71] Sven Heiberg and Jan Willemson. Verifiable Internet Voting in Estonia. In *Proceedings of the 6th Conference on Electronic Voting*, pages 23–30, 2014.

[72] I D Hill, B A Wichmann, and D R Woodall. Single Transferable Vote by Meek's Method. *The Computer Journal*, 30(3):277–281, 1987.

[73] Veronika Hinz and Markku Suksi. The Electoral Cycle: On the Right to Participate in the Electoral Process. In *Election Elements: On the International Standards of Electoral Participation*, pages 1–42. 2003.

[74] Gerard J Holzmann. The Model Checker SPIN. *Ieee Transactions on Software Engineering*, 23(5):279–295, 1997.

[75] Michael Huth and Mark Ryan. *Logic in Computer Science*, volume 1. Cambridge University Press, Cambridge, 2004.

[76] IBM. NCSA Common log format, 2004.

[77] CADE Inc. CADE Bylaws (effective Nov. 1, 1996; amended July/August 2000), 2000.

[78] International IDEA. What is the Electoral Cycle.

[79] Franjo Ivančić, Zijiang Yang, Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Efficient SAT-based bounded model checking for software verification. *Theoretical Computer Science*, 404(3):256–274, 2008.

[80] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, 2006.

[81] Simon Peyton Jones. Haskell 98 Language and Libraries – The Revised Report. *Journal of Functional Programming*, 2003.

[82] Ari Juels, Dario Catalano, and Markus Jakobsson. Coercion-Resistant Electronic Elections. *Proc. workshop on Privacy in the electronic societ*, pages 61–70, 2005.

[83] Carmen Kempka. Coercion-Resistant Electronic Elections with Write-In Candidates. *Electronic Voting Technology Workshop*, 2012.

[84] Karen Kent and Murugiah Souppaya. Guide to Computer Security Log Management. *National Institute of Standards and Technology*, 2006.

[85] Joseph R. Kiniry, Dermot Cochran, and Patrick E. Tierney. Verification-Centric Realization of Electronic Vote Counting. *Electronic Voting Technology Workshop*, 2007.

[86] Steve Kremer, Mark Ryan, and Ben Smyth. Election Verifiability in Electronic Voting Protocols.

[87] Robert Krimmer, Stefan Triessnig, and Melanie Volkamer. The development of remote e-voting around the world: A review of roads and directions. *E-Voting and Identity*, pages 1–15, 2007.

[88] Leslie Lamport. Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers [Book Review]. *Computer*, 35(9):81–81, 2002.

[89] John D. C. Little. A Proof for the Queuing Formula: L = λ W, 1961.

[90] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*, volume 53. Springer New York, New York, NY, 1992.

[91] Nicolas Markey. Temporal logic with past is exponentially more succinct. *EATCS Bulletin*, 79(2):122–128, 2003.

[92] M. A. McGaley. Electronic Voting: A Safety Critical System. 2003.

[93] Margaret Mcgaley. *E-voting: an Immature Technology in a Critical Context*. PhD thesis, National University of Ireland, Maynooth, 2008.

[94] Natarajan Meghanathan. Source Code Analysis to Remove Security Vulnerabilities in Java Socket Programs: A Case Study. *International Journal of Network Security and Its Applications*, 5(1):1–16, 2013.

[95] Tommi Meskanen and Hannu Nurmi. Closeness Counts in Social Choice. In *Power, Freedom, and Voting*, pages 289–306. Springer Berlin Heidelberg, Berlin, Heidelberg.

[96] Laurent D. Michel, Alexander A. Shvartsman, and Nikolaj Volgushev. A Systematic Approach to Analyzing Voting Terminal Event Logs. *USENIX Journal of Election Technology and Systems (JETS)*, 2(2):34–53, 2014.

[97] Motorola Solutions. Analysis of IOS 8 MAC Randomization on Locationing. Technical Report October, 2014.

[98] Eric Pacuit. Voting Methods, 2012.

[99] David A. Plaisted. A Consideration of the New CADE Bylaws.

[100] NIST Special Publications. *Generally Accepted Principles and Practices for Securing Information Technology Systems.* Number September. 1996.

[101] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. pages 337–351. 1982.

[102] A Riera and J Borrell. Practical Approach to Anonymity in Large Scale Electronic Voting Schemes. *{NDSS'99}, Network and Distributed System Security Symposium*, pages 69–82, 1999.

[103] Muriel Roger and Jean Goubault-Larrecq. Log Auditing through Model Checking. In *Proc. 14th IEEE Computer Security Foundations Workshop (CSFW'01)*, number June, pages 220–236, 2001.

[104] Carsten Schürmann and Jian Wang. Measuring Voter Lines. Technical Report November, IT University of Copenhagen, 2015.

[105] Roberto Sebastiani and Stefano Tonetta. "More Deterministic" vs. "Smaller" Büchi Automata for Efficient LTL Model Checking. *Correct Hardware Design and Verification ...*, pages 126–140, 2003.

[106] Ben Smyth, Mark Ryan, Steve Kremer, and Mounira Kourjieh. Towards automatic analysis of election verifiability properties. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6186 LNCS:146–163, 2010.

[107] Fabio Somenzi and Roderick Bloem. Efficient Buchi Automata from LTL Formulae. *Computer Aided Verification*, (1855):1–17, 2000.

[108] Douglas M Spencer and Zachary S. Markovits. Long Lines at Polling Stations? Observations from an Election Day Field Study. *Election Law Journal*, 9(1):3–17, 2010.

[109] Charles Stewart-III. Managing Polling Place Resources. Technical Report November, Caltech/MIT Voting Technology Project, 2015.

[110] Yanjie Sun, Chenyi Zhang, Jun Pang, Baptiste Alcalde, and Sjouke Mauw. A trust-augmented voting scheme for collaborative privacy management. *Journal of Computer Security*, 20(4):437–459, 2012.

[111] Thomas Tjøstheim, Thea Peacock, and Peter Y. A. Ryan. A model for system-based analysis of voting systems. *Security Protocols*, pages 1–19, 2010.

[112] Wil M. P. van der Aalst. *Process Mining: Discovery, Conformance and Enhancement of Business Processes.* Springer, 2011.

[113] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[114] Wikipedia. Monotonicity Criterion.

[115] Wikipedia. Single Transferable Vote.

[116] Pengfei Zhou, Yuanqing Zheng, and Mo Li. How Long to Wait?: Predicting Bus Arrival Time with Mobile Phone based Participatory Sensing. *Proceedings of the 10th international conference on Mobile systems, applications, and services - MobiSys '12*, 13(6):379–392, 2012.

Appendix A

Appendix

A.1 Additional Example of SMT Solver: Norwegian and Australian Election

We continue the study of election systems from Norway and Victoria state of Australia discussed in Chapter 3. In this section, We show how these properties presented in Chapter 3 can also be formulated with the method for voting scheme analysis, which provides us another way of checking these properties.

A.1.1 Norwegian Election

The related events in the logs of the vote collector server and the cleanser are encoded into arrays.

1. *Ballot* denotes the ballot array from the vote collector server, where each $Ballot(i) := (id, v)$ is the tuple of stored ballot id and voter id;

2. *Read* denotes the read event array from the cleanser, where each $Read(i) := (id, v)$ is the read-in ballot id and voter id;

3. *Accept* denotes the accept event array, where each $Accept(i)$ is the accepted ballot id;

4. *Reject* denotes the reject event array, where each $Reject(i)$ is the rejected ballot id.

We use $|A|$ to represent the length of the array A. We define $(id, v) = (id', v')$ if and only if $id = id'$ and $v = v'$. And also define $(id, v)[1] := id$ and $(id, v)[2] := v$.

Policy 1: The vote collector and the cleanser agree on which ballots actually exist. The formula states that each ballot must be read by the cleanser and conversely that every ballot in the cleanser needs to be in vote collector server. Furthermore since the order of ballots matters, we wish to make sure that the ordering of any two ballots is preserved between vote collector server and cleanser, where as in Chapter 3 we separate the two requirements into two formulas.

$$(|Ballot| = |Read|) \wedge (\forall i.(1 \leq i \leq |Ballot|) \rightarrow Ballot[i] = Read[i]))$$

Policy 2: Every read ballot must either be accepted or rejected later.

$$\forall i.(1 \leq i \leq |Read|) \rightarrow (\exists j.(Accept[j] = Read[i][1]) \vee (Reject[j] = Read[i][1]))$$

Policy 3: A ballot should not both be rejected and accepted.

$$\forall i.(1 \leq i \leq |Accept|) \rightarrow (\forall j.(1 \leq j \leq |Reject|) \rightarrow (Accept[i] \neq Reject[i]))$$

Policy 4: The next policy expresses only the last vote cast should be accepted. The policy requires that all earlier ballots from the same voter must be rejected.

$$\forall i, j.(1 \leq i < j \leq |Read| \wedge Read[i][2] = Read[j][2]) \rightarrow \exists k.Reject[k] = Read[i][1]$$

Policy 5: For every voter who voted, at least one ballot is accepted.

$$\forall i.(1 \leq i \leq |Ballot|) \rightarrow \exists j, k.Ballot[i][2] = Read[j][2] \wedge Ballot[j][1] = Accept[k]$$

A.1.2 Victoria State Election

In the same way as the Norwegian example, we encode the Victoria State of Australia election logs from different agents into arrays. Each ballot's serial number is unique. The agents are client VPS, client EMV and the 5 MBB peers. We show here that by including temporal order (time stamps) in the array, temporal properties can be expressed without temporal operators.

1. *RecExt* denotes the receive external message event, where $RecExt[i] := (t, id, ty)$, t is the time, id is the ballot id and ty is the message type;

2. *SendM* denotes the send message event from either VPS($SendM[1]$) or EVM ($SendM[2]$), where $Send[i][j] := (t, id, ty)$ is defined same as *RecExt*;

3. *RecRes* denotes the client (VPS, $RecRes[1]$ or EVM, $RecRes[2]$) that receives the confirmation from the peers, where $RecRes[i][j] := (t, id, peer, ty)$, t is the time, id is the ballot id, $peer$ is the MBB peer $\{1...5\}$, ty is the response message type;

4. *ValidSig* denotes the client (VPS, $ValidSig[1]$ or EVM, $ValidSig[2]$) that validates the signature, where $ValidSig[i][j] := (t, id, peer)$ is the same as defined above;

5. *MetThld* denotes the client (VPS, $MetThld[1]$ or EVM, $MetThld[2]$) that check if that threshold is met, where $MetThld[i][j] := (t, id)$ is the same as defined above, t is the time, and id is the ballot id.

Policy 1: The first policy we show is that when a peer receives an external messages, it must have originated from either from VPS or EVM. $t' < t$ means time t' is earlier than t.

$$\begin{gathered}\forall i.(1 \leq i \leq 5) \rightarrow \\ \forall (t, id, ty) \in RecExt[i]. \exists (t', id, ty) \in SendM[1] \cup SendM[2]. t' < t\end{gathered}$$

Policy 2: The second policy that we check is that the threshold is only met if it is preceded by at least four (out of five) valid signature checks from different peers. Formally we check that at least one of the five possible scenarios are satisfied which is what the formula φ does.

$$\begin{aligned}
&\text{Let } \psi(t, id, c, \overline{ps}) &&= \textstyle\bigwedge_{p \in \overline{ps}} \exists (t', id, p) \in ValidSig[c]. t' < t \\
&\text{Let } peers &&= \{1, 2, 3, 4, 5\} \\
&\text{Let } \varphi(c) &&= \forall (t, id) \in MetThld[c]. \textstyle\bigvee_{p \in peers} \psi(t, id, c, peers \setminus \{p\})
\end{aligned}$$

We check this policy, φ, for every client, i.e for both VPS and EVM.

$$\varphi(1) \ \wedge \ \varphi(2)$$

Policy 3: If a client validates a response from a peer, it should have first received a response from that peer. First we define formula $\varphi'(c)$ for one client c.

$$\forall (t, id, peer) \in ValidSig[c]. \exists (t', id, peer, ty) \in RecRes[c]. t' < t$$

We check this policy, φ', for every client VPS and EVM in a way similar to before.

$$\varphi'(1) \ \wedge \ \varphi'(2)$$

Policy 4: Any ballot that was audited on peer i, it should not be used later for voting. If such a case occurs EVM should have logged "*error*".

$$\begin{aligned}&\forall(t, id, \text{"}audit\text{"}) \in RecExt[i].\\ &\ (\exists(t', id, \text{"}startevm\text{"}) \in SendM[2] \wedge t' > t\\ &\ \to \exists(t'', id, peer, \text{"}error\text{"}) \in RecRes[2] \wedge t'' > t')\end{aligned}$$

Policy 5: A voter cannot reuse a ballot, which will also incur a logged error message by EVM.

$$\begin{aligned}\forall(t, id, \text{"}startevm\text{"}) \in sendM[2].(\exists(t', id, \text{"}startevm\text{"}) \in SendM[2] \wedge t' > t \to\\ \exists(t'', id, peer, \text{"}error\text{"}) \in RecRes[2] \wedge t'' > t')\end{aligned}$$

A.2 Device Appearance

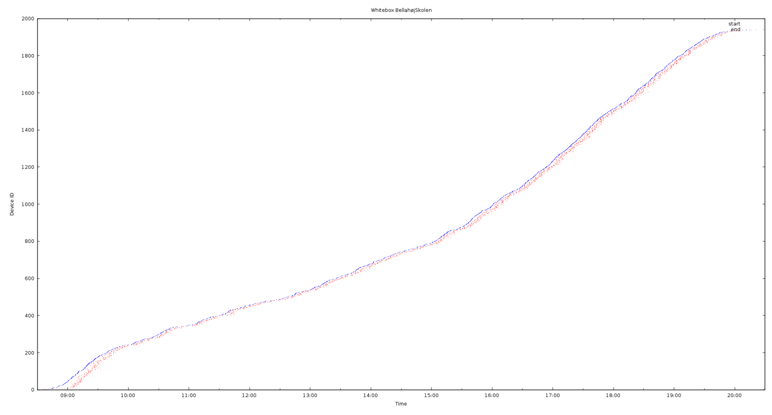


Figure A.1: Device Appearance in Bellahøj Skole Copenhagen

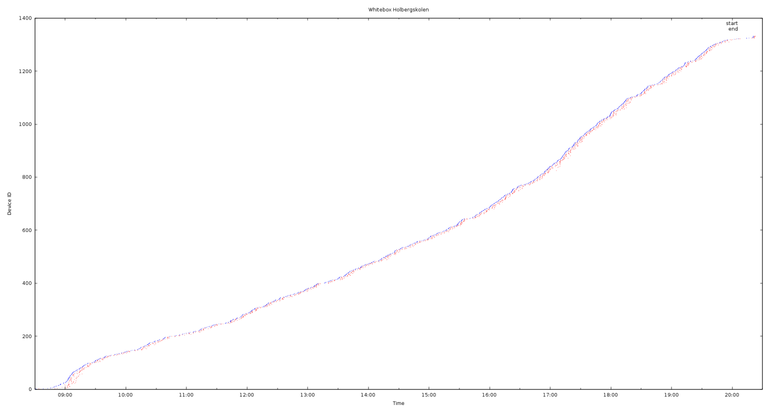


Figure A.2: Device Appearance in Holbergskolen Copenhagen

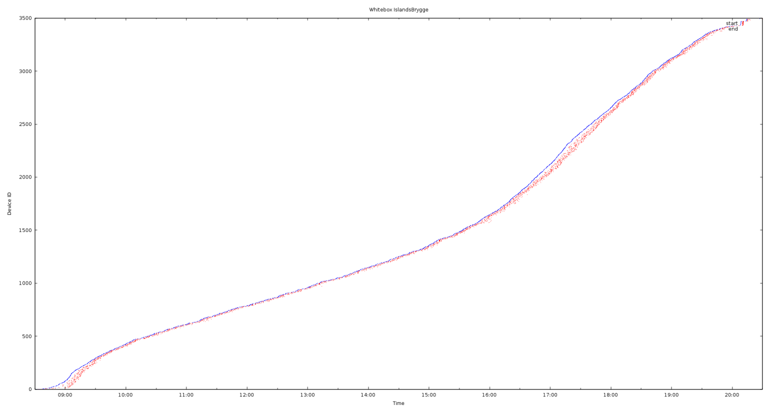


Figure A.3: Device Appearance in Islands Brygge Skolen Copenhagen

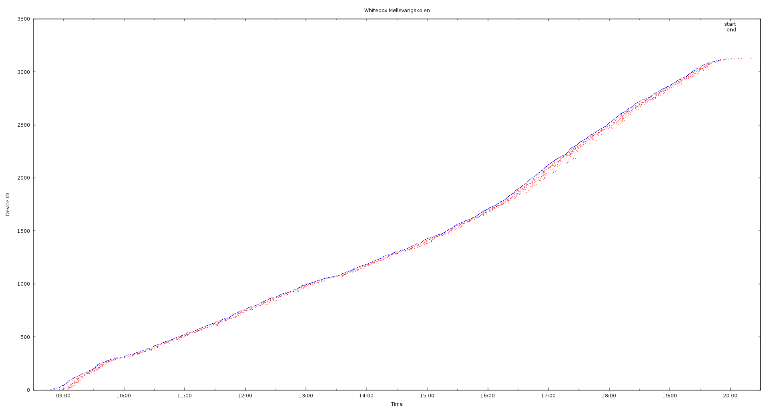


Figure A.4: Device Appearance in Møllevangskolen Aarhus

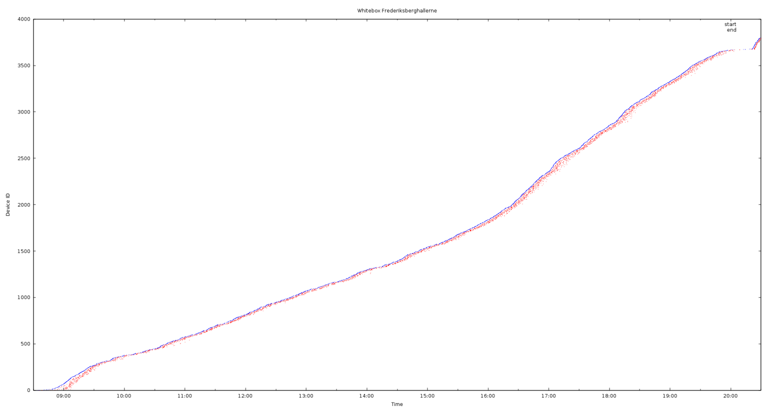


Figure A.5: Device Appearance in Frederiksbjerg Hallerne Aarhus

A.3 Average Waiting Time via Averaging Method

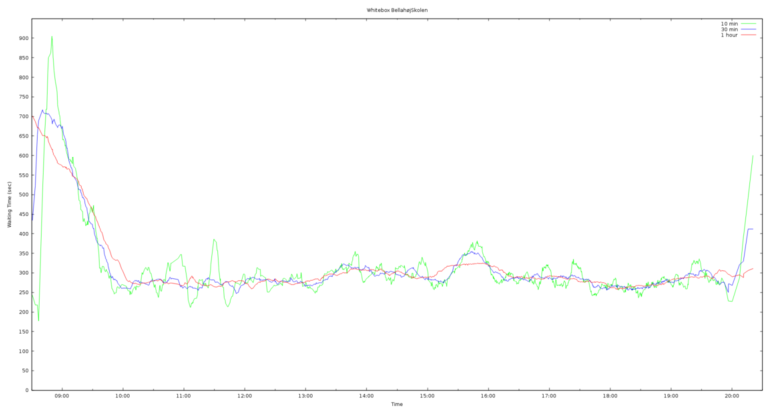


Figure A.6: Average Time (Averaging Method) in Bellahøj Skole Copenhagen

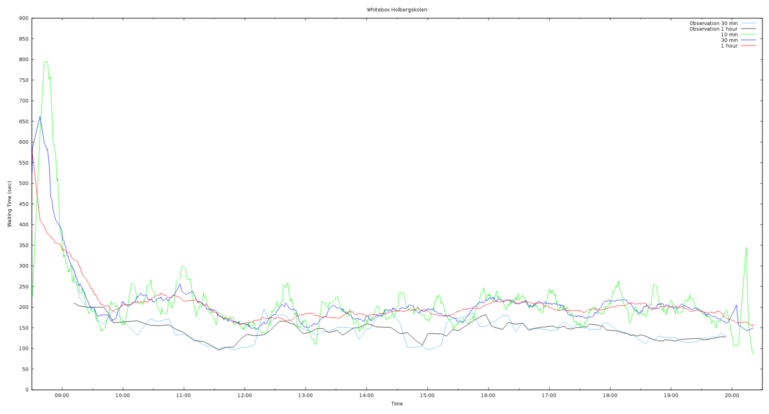


Figure A.7: Average Time (Averaging Method) in Holbergskolen Copenhagen

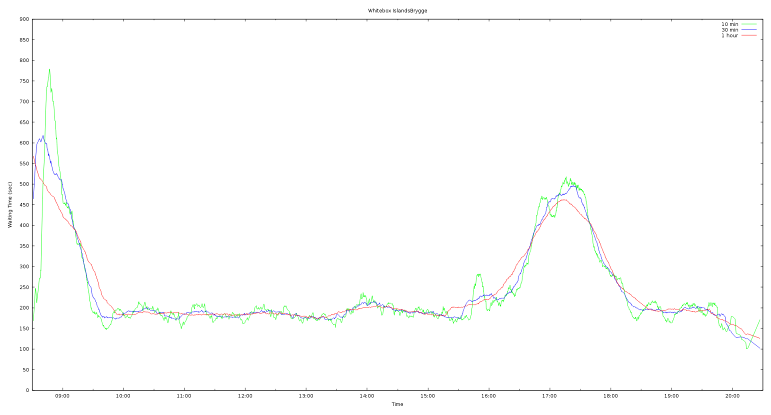


Figure A.8: Average Time (Averaging Method) in Islands Brygge Skolen Copenhagen

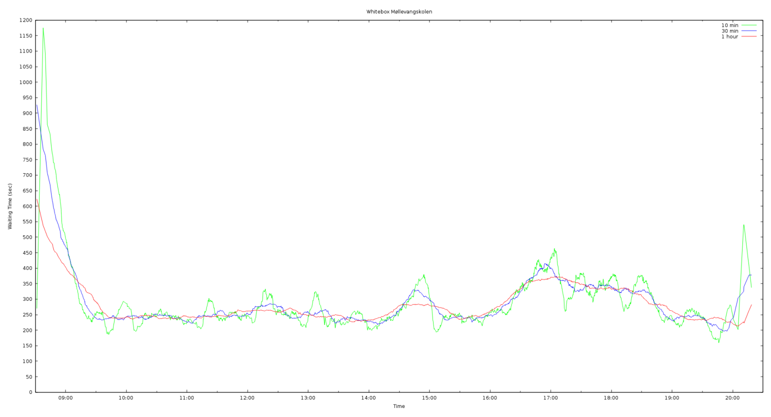


Figure A.9: Average Time (Averaging Method) in Møllevangskolen Aarhus

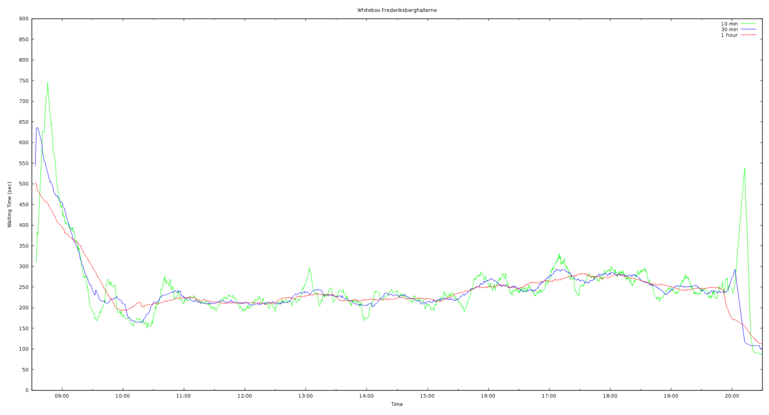


Figure A.10: Average Time (Averaging Method) in Frederiksbjerg Hallerne Aarhus

A.4 Average Waiting Time via Little's Method

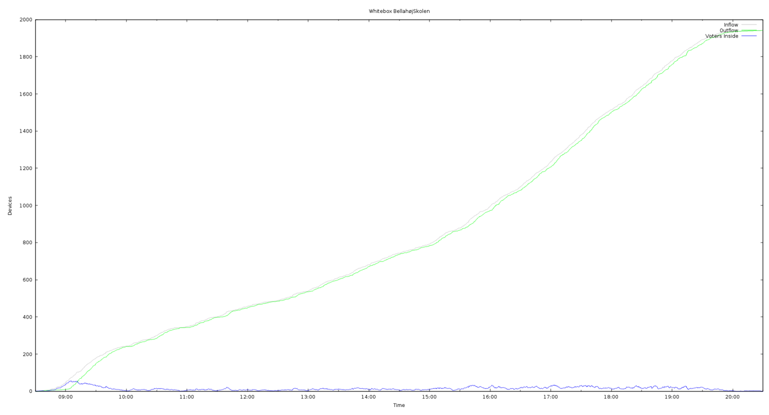


Figure A.11: Device Flow in Bellahøj Skole Copenhagen

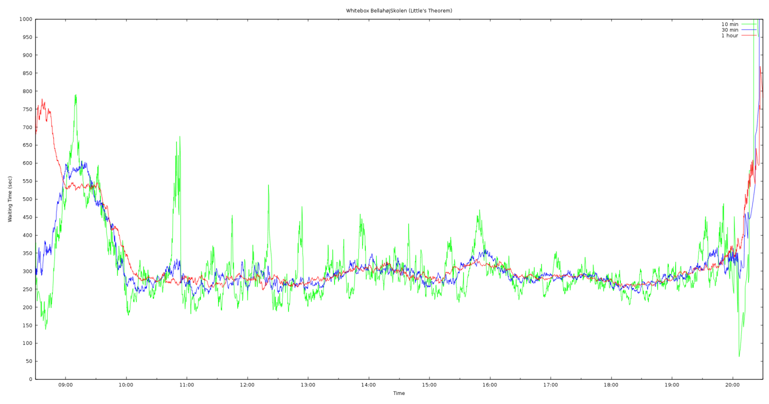


Figure A.12: Average Time (Little's Method) in Bellahøj Skole Copenhagen

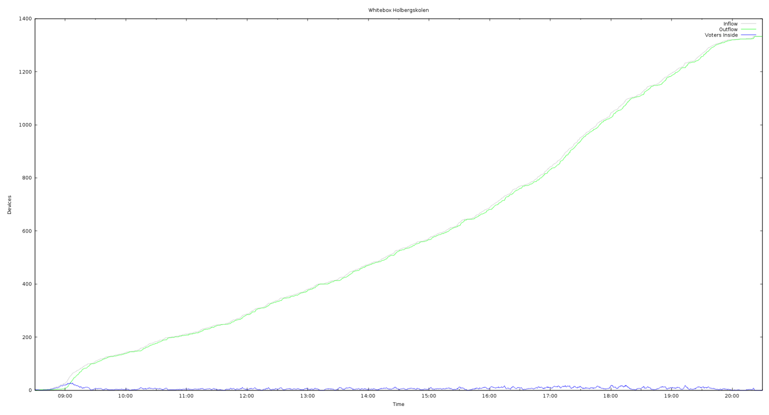


Figure A.13: Device Flow in Holbergskolen Copenhagen

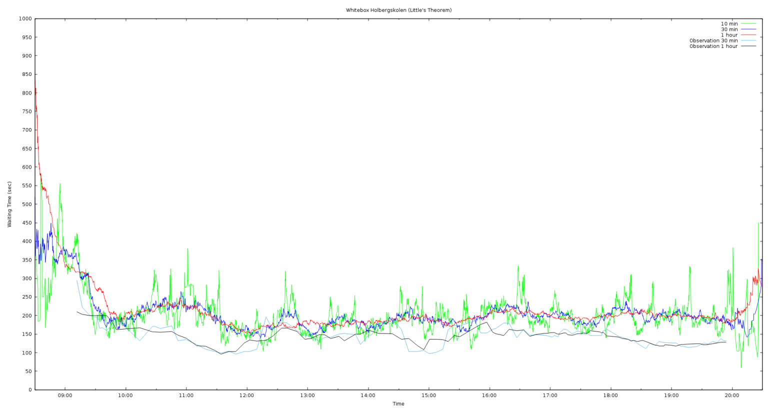


Figure A.14: Average Time (Little's Method) in Holbergskolen Copenhagen

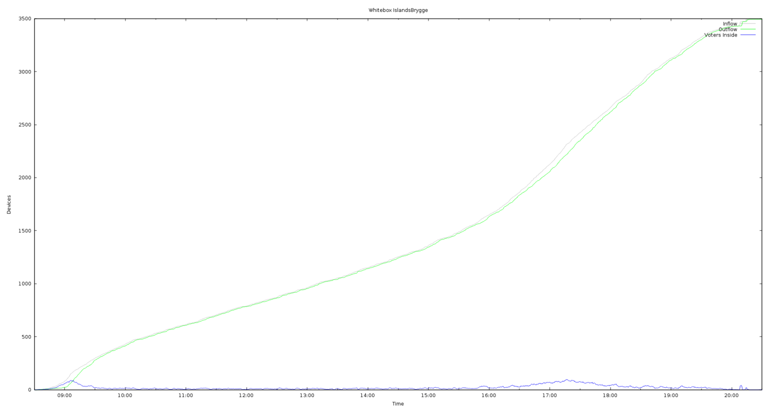


Figure A.15: Device Flow in Islands Brygge Skolen Copenhagen

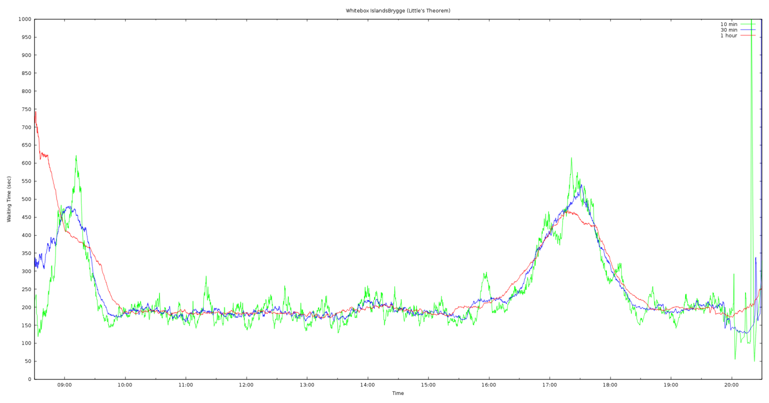


Figure A.16: Average Time (Little's Method) in Islands Brygge Skolen Copenhagen

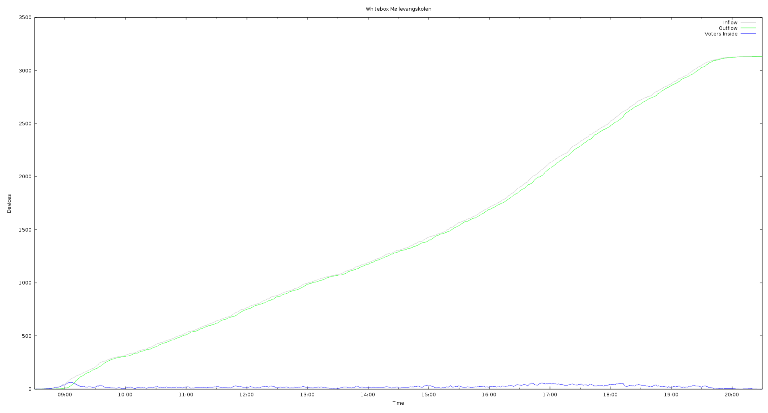


Figure A.17: Device Flow in Møllevangskolen Aarhus

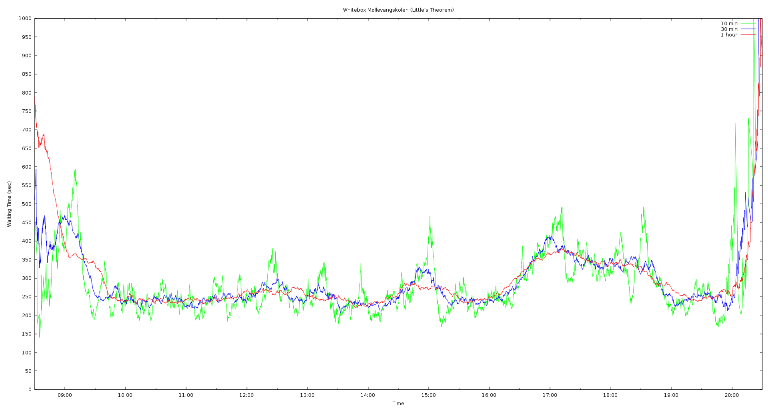


Figure A.18: Average Time (Little's Method) in Møllevangskolen Aarhus

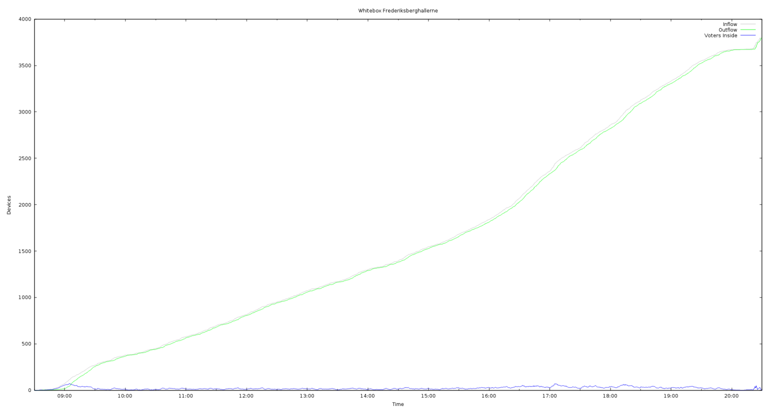


Figure A.19: Device Flow in Frederiksbjerg Hallerne Aarhus

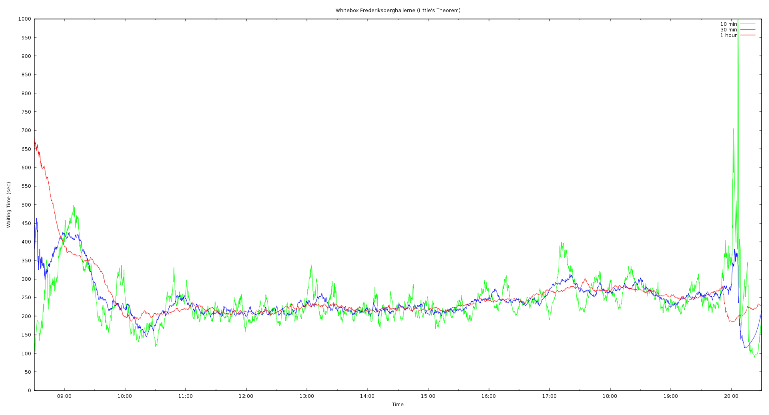


Figure A.20: Average Time (Little's Method) in Frederiksbjerg Hallerne Aarhus