Formal Methods for Trustworthy Voting Systems

From Trusted Components to Reliable Software

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der KIT-Fakultät für Informatik

des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Michael Kirsten

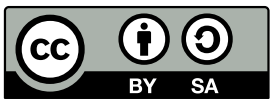
aus Dresden

Tag der mündlichen Prüfung: 26. Januar 2022

Referiert von:

1. Prof. Dr. rer. nat. Bernhard Beckert
   Karlsruher Institut für Technologie (KIT)
   Deutschland

2. Prof. Carsten Schürmann, Ph. D.
   IT-Universitetet i København (ITU)
   Dänemark



Michael Kirsten

Formal Methods for Trustworthy Voting Systems

From Trusted Components to Reliable Software

Acknowledgements

Still experiencing a slight sense of disbelief—nonetheless immense delight—that I somehow ended up finishing the writing of a doctoral dissertation, I am happy that I hereby get the opportunity to acknowledge and express my appreciation to some of the people without whom this project would not have been possible and who made the whole journey worthwhile. A dissertation is not written by one person alone and—with the risk of sounding cheesy—I cannot thank and mention all the many people who were involved (in some way or another) in its process by name. Hence, I wish to start by thanking everyone who supported me in this adventure! In the following, I would like to address some of them explicitly.

Foremost, I would like to express my deep gratitude to my supervisor Bernhard Beckert for both sparking my interest in formal logic and formal methods in his lectures in the first place, and for giving me the opportunity, encouragement, and guidance to pursue student research projects, my diploma thesis, and finally a PhD in his group. I wish to thank Bernhard for lots of useful advice and inspiration throughout, productive and motivating discussions in the process of some contributions to the results presented in this thesis, and a great deal of patience and freedom to pursue the ideas and collaborations of my choice without pressure or worries.

I am also very grateful to Carsten Schürmann both for agreeing without any hesitation to act as my second referee—while also providing lots of patience—and for the great collaboration during my visit in Copenhagen with a warm welcome in his group and the fruitful cooperation on the subsequent publication. Thanks go also to Hannes Hartenstein for his feedback on an early presentation of this thesis and for agreeing to act as examiner with great cooperation on making it easy to find a date for my defense. I further appreciate the effort and helpful feedback from all the anonymous reviewers on improving the publications of the presented results. Additional proofreading of parts of this thesis was thankfully provided by Jonas Schiffl.

Moreover, I thank all my collaborators for their contributions to the presented results and the pleasant, inspiring and fruitful discussions during the process. In particular, I thank Thorsten Bormer, Olivier Cailloux, Rajeev Goré, Daniel Grahl, Mihai Herda, Vladimir Klebanov, Alexander Koch, Tomasz Truderung, Shmuel Tyszberowicz and Mattias Ulbrich for the pleasant and rewarding discussions that—besides being inspiring, providing helpful feedback, and supporting my research—made the whole process worthwhile. I would also like to thank the many students who I had the pleasure of supervising or co-supervising over the years, where some of them contributed to significant parts of this thesis either by helping me shape my ideas, or even as co-authors, co-developers, or co-proof-engineers. Especially, I thank Stephan Bohr, Karsten Diekhoff, Jonas Klamroth,

Holger Klein, Jonas Krämer, Jörn Kußmaul, Florian Lanzinger, Till Neuber, Fabian Richter, Jonas Schiffl, Michael Schrempp, Lukas Stapelbroek, and Marion Steinriede.

Many thanks go to all of my current and former colleagues—especially the ones I had the pleasure to collaborate with in interesting research ideas and thesis supervisions—for the nice working atmosphere, and great and fruitful discussions on- and off-topic, as they always found the time for interesting and inspiring discussions. Notably, I wish to thank Ralf Kölmel for his great support and patience with special requests and rescuing my computer system for yet another time, Simone Meinhart for her support and patience with all administrative tasks, helping me to stay focused and productive when most needed, as well as Alexandra Andresh and Audrey Bohlinger for their great support in all the formalities. Special thanks go to Tianhai Liu for being a great office mate, Sarah Grebing for the inspiring cooperation on research-oriented teaching, and Lionel Blatter, Daniel Grahl, Simon Greiner, and Mihai Herda for a lot of helpful feedback and many motivating off-and on-topic discussions during lunch.

Furthermore, I would like to thank the COST Action IC1205, the German Research Foundation (DFG), and the German Federal Ministry of Education and Research (BMBF) for financing my work. I am also very grateful for the motivating words and great atmosphere during this journey by all the friends and long-lasting acquaintances that I made on project cooperations, workshops, conferences, and summer and winter schools. Last but not least, I wish to thank my family and friends for all the understanding, trust, and support and many helpful and joyful distractions over the years of this endeavor.

Abstract

VOTING is prominently an important part of democratic societies, and its outcome may have a dramatic and broad impact on societal progress. Therefore, it is paramount that such a society has extensive trust in the electoral process, such that the system's functioning is reliable and stable with respect to the expectations within society. Yet, with or without the use of modern technology, voting is full of algorithmic and security challenges, and the failure to address these challenges in a controlled manner may produce fundamental flaws in the voting system and potentially undermine critical societal aspects.

In this thesis, we argue for a development process of voting systems that is rooted in and assisted by formal methods that produce transparently checkable evidence for the guarantees that the final system should provide so that it can be deemed trustworthy. The goal of this thesis is to advance the state of the art in formal methods that allow to systematically develop trustworthy voting systems that can be provenly verified. In the literature, voting systems are modeled in the following four comparatively separable and distinguishable layers: (1) the physical layer, (2) the computational layer, (3) the election layer, and (4) the human layer. Current research usually either mostly stays within one of those layers or lacks machine-checkable evidence, and consequently, trusted and understandable criteria often lack formally proven and checkable guarantees on software-level and vice versa.

The contributions in this work are formal methods that fill in the trust gap between the principal *election layer* and the *computational layer* by a reliable translation of trusted and understandable criteria into trustworthy software. Thereby, we enable that executable procedures can be formally traced back and understood by election experts without the need for inspection on code level, and trust can be preserved to the trustworthy system.

The works in this thesis all contribute to this end and consist in five distinct contributions, which are the following:

(I) a method for the generation of secure card-based communication schemes,

(II) a method for the synthesis of reliable tallying procedures,

(III) a method for the efficient verification of reliable tallying procedures,

(IV) a method for the computation of dependable election margins for reliable audits,

(V) a case study about the security verification of the GI voter-anonymization software.

These contributions span formal methods on illustrative examples for each of the three principal components, (1) voter-ballot box communication, (2) election method, and (3) election management, between the election layer and the computational layer.

Within the first component, the voter-ballot box communication channel, we build a bridge from the communication channel to the cryptography scheme by automatically generating secure card-based schemes from a small formal model with a parameterization of the desired security requirements. For the second component, the election method, we build a bridge from the election method to the tallying procedure by (1) automatically synthesizing a runnable tallying procedure from the desired requirements given as properties that capture the desired intuitions or regulations of fairness considerations, (2) automatically generating either comprehensible arguments or bounded proofs to compare tallying procedures based on user-definable fairness properties, and (3) automatically computing concrete election margins for a given tallying procedure, the collected ballots, and the computed election result, that enable efficient election audits. Finally, for the third and final component, the election management system, we perform a case study and apply state-of-the-art verification technology to a real-world e-voting system that has been used for the annual elections of the German Informatics Society (GI – "Gesellschaft für Informatik") in 2019. The case study consists in the formal implementation-level security verification that the voter identities are securely anonymized and the voters' passwords cannot be leaked.

The presented methods assist the systematic development and verification of provenly trustworthy voting systems across traditional layers, i.e., from the election layer to the computational layer. They all pursue the goal of making voting systems trustworthy by reliable and explainable formal requirements. We evaluate the devised methods on minimal card-based protocols that compute a secure AND function for two different decks of cards, a classical knock-out tournament and several Condorcet rules, various plurality, scoring, and Condorcet rules from the literature, the Danish national parliamentary elections in 2015, and a state-of-the-art electronic voting system that is used for the German Informatics Society's annual elections in 2019 and following.

Zusammenfassung

Wahlen sind bekannterweise ein wichtiges Element in demokratischen Gesellschaften und ihr Ausgang hat potentiell einen dramatischen und weitreichenden Einfluss auf den Fortschritt der Gesellschaft. Aus diesem Grund ist es immanent wichtig, dass der Wahlvorgang umfangreiches Vertrauen der Gesellschaft genießt, damit die Funktionsweise des Systems verlässlich und stabil die gesellschaftlichen Erwartungen erfüllt. Wählen ist jedoch, mit oder ohne Unterstützung moderner Technologien, reich an algorithmischen und sicherheitstechnischen Herausforderungen und ohne einen kontrollierten Umgang damit führen diese zu womöglich fundamentalen Defekten des Wahlsystems, mit dem Risiko, dass kritische Aspekte der Gesellschaft untergraben werden.

In dieser Arbeit sprechen wir uns für einen Entwicklungsprozess von Wahlsystemen aus, der auf formalen Methoden und deren Unterstützung aufbaut, die transparent überprüfbare Evidenz für die Garantien des finalen Systems liefern, damit dieses als vertrauenswürdig erachtet werden kann. Das Ziel dieser Arbeit ist die Weiterentwicklung vom Stand der Technik formaler Methoden, die es ermöglichen, glaubwürdige Wahlsysteme, die mithilfe von Beweisen überprüft werden können, systematisch zu entwickeln. In der Literatur werden Wahlsysteme in den folgenden vergleichsweise separierten und unterscheidbaren Schichten modelliert: (1) die physikalische Schicht, (2) die rechenintensive Schicht, (3) die Wahlschicht und (4) die menschliche Schicht. Aktuelle Forschung bleibt für gewöhnlich meist innerhalb einer dieser Schichten oder ihr mangelt es an maschinell überprüfbarer Evidenz, sodass zuverlässigen und verständlichen Kriterien oft die formal bewiesenen und prüfbaren Garantien fehlen und umgekehrt.

Der Beitrag dieser Arbeit sind formale Methoden die die Vertrauenslücke zwischen der hauptsächlichen *Wahlschicht* und der *rechenintensiven Schicht* ausfüllen, indem sie

eine zuverlässige Übersetzung vertrauensbehafteter und verständlicher Kriterien zu glaubwürdiger Software liefern. Damit ermöglichen wir ausführbare Prozeduren, die von Wahlexpertinnen oder Wahlexperten formal zurückverfolgt und verstanden werden können, ohne dabei den Programmcode durchsehen zu müssen, sodass das Vertrauen bis zum glaubwürdigen System erhalten werden kann.

Die Arbeiten dieser Dissertation tragen dabei alle zu diesem Ziel bei und bestehen aus den folgenden fünf unterschiedlichen Beiträgen:

(I) eine Methode zur Generierung sicherer spielkartenbasierter Kommunikationsprotokolle,

(II) eine Methode zur Synthese verlässlicher Auszählverfahren,

(III) eine Methode zur effizienten Verifikation verlässlicher Auszählverfahren,

(IV) eine Methode zur Berechnung zuverlässiger Gewinnmargen bei Wahlen zur Durchführung verlässlicher Audits,

(V) eine Fallstudie zur Verifikation der Sicherheit der Anonymisierung von Wählenden innerhalb des E-Voting-Systems bei der Gesellschaft für Informatik.

Diese Beiträge umfassen formale Methoden anhand illustrativer Beispiele für jede der folgenden drei grundsätzlichen Komponenten, (1) dem Kommunikationskanal zwischen Wählenden und der Wahlurne, (2) der Wahlmethode und (3) dem Wahlmanagementsystem zwischen der Wahl- und der rechenintensiven Schicht.

Innerhalb der ersten Komponente, dem Kommunikationskanal zwischen Wählenden und Wahlurne, bauen wir eine Brücke vom Kommunikationskanal zum kryptographischen Protokoll, indem wir sichere spielkartenbasierte Protokolle automatisch generieren, basierend auf einem kleinen formalen Modell, das in den gewünschten Sicherheitseigenschaften parametrisierbar ist. Für die zweite Komponente, die Wahlmethode, bauen wir eine Brücke von der Wahlmethode hin zum ausführbaren Auszählverfahren, indem wir (1) die ausführbare Prozedur für die gewünschten Anforderungen, in Form von Eigenschaften, die angestrebte Intuitionen oder Regulierungen von Fairnessbetrachtungen abbilden, automatisch synthetisieren, (2) automatisch entweder verständliche Argumente oder beschränkte Beweise generieren, mithilfe derer sich Wahlverfahren basierend auf von den Anwendenden definierbaren Fairnesseigenschaften und Auszählverfahren vergleichen lassen, sowie (3) automatisch konkrete Gewinnspannen einer Wahl für ein gegebenes Auszählverfahren, abgegebene Stimmzettel, sowie das berechnete Wahlergebnis berechnen, um effiziente Wahl-Audits zu ermöglichen. Abschließend führen wir für die dritte und letzte Komponente, das Wahlmanagementsystem, eine Fallstudie durch, bei der wir Verifikationstechnologien nach dem Stand der Technik auf ein real eingesetztes E-Voting-System anwenden, das für die jährlichen Wahlen der deutschen Gesellschaft für Informatik (GI) eingesetzt wurde. Die Fallstudie besteht aus der formalen Verifikation

auf Implementierungsebene von einer Sicherheitseigenschaft, die die sichere Anonymisierung der Identitäten aller Wählenden sicherstellt, damit deren Passwörter nicht abfließen können.

Die vorgestellten Methoden unterstützen die systematische Entwicklung und Verifikation von bewiesen glaubwürdigen Wahlsystemen über traditionelle Schichten hinweg, das heißt von der, beispielsweise durch Regulierungen definierten, Wahlschicht hin zu ausführbaren Prozeduren. Die Methoden verfolgen alle das Ziel eines Entwicklungsprozesses glaubwürdiger Wahlsysteme anhand verlässlicher und erklärbarer formaler Anforderungen. Wir evaluieren die entworfenen Methoden für karten- und lauf-minimale spielkartenbasierte Protokolle, die eine sichere UND-Funktion für zwei verschiedene Kartensätze berechnen, ein klassisches Knock-Out-Turnier und verschiedene Condorcet-Methoden, verschiedene einfache Mehrheitswahlverfahren, punktebasierte Verfahren und Condorcet-Methoden aus der Literatur, die dänischen Parlamentswahlen 2015, sowie ein E-Voting-System nach dem Stand der Technik, das für die jährlichen Wahlen der deutschen Gesellschaft für Informatik (GI) genutzt wird.

Contents

List of Definitions and Theorems

List of Figures

Page

List of Tables

List of Algorithms

List of Listings

Part I

Introduction and Foundations

"Begin at the beginning," the King said, gravely, "and go on till you come to the end: then stop."

Lewis Carroll, *Alice in Wonderland*, 1865

1 Introduction

> An original idea. That can't be too hard. The library must be full of them.
>
> Stephen Fry, *The Liar*, 1991

The task of voting is prominently an important part of – and may have a dramatic and broad impact on – a democratic society. Therefore, fundamental flaws in a voting system carry the potential of undermining critical societal aspects, especially when much is at stake and conflicting interests arise, e.g., for electing a president or a parliament. Balancing such interests – or requirements – for a reliable and acceptable election outcome may generally be impossible in a strict sense, and finding a viable and still reliable tradeoff is very often non-trivial, cumbersome, and error-prone. Hence, the development of trustworthy voting systems is full of pitfalls that all potentially compromise or harm the reliability of the whole system. Consequently, we argue for a development that is rooted in and assisted by formal methods that produce transparently checkable evidence for the guarantees that the final system provides so that it can be deemed trustworthy.

Albeit this thesis targets specifically voting systems, the notions and definitions that we use to model voting systems throughout this thesis span – to a large extent – the general task of determining a collective decision from a set of individual choices. The task of determining a collective decision from a set of individual choices is part of many systems, e.g., for computations on large sets of data, in a distributed setting, or when multiple agents are involved (see, e.g., the positions by Maggs and Sitaraman (2015)). With increasing computing capacities, such aggregation tasks are often automated by algorithms coined as some form of artificial intelligence (AI) and used ubiquitously,

e.g., when calculating optimal prices for a flight, preparing online recommendations or search results, and in various other scenarios. Whether the intelligence is artificial or not, such a task commonly involves a plethora of algorithmic or security challenges to be addressed for the system to work reliably, especially when dealing with critical or personal data, or when critical decisions depend on the computed result. Within modern technology, such a system might be, e.g., an AI agent that provides personalized services or products, a method that selects the best output from multiple specialized machine learning classifiers (Cornelio et al., 2021; Conitzer and Sandholm, 2012), or a distributed ledger (DLT) system that computes a (collectively) robust (block-)chain of transactions (Sompolinsky, Wyborski, and Zohar, 2018). For the matter of this thesis, we consider the task of making a single collective decision from multiple individual preferences via a *voting system*. Voting, from an abstract point of view, subsumes many of the aforementioned scenarios and is moreover, in concrete terms, clearly an important and critical application in its own right, say when a group of people (the voters) wants to elect one (or a limited amount) of many alternatives.

1.1 Objective

In this thesis, we devise and evaluate targeted formal methods which, for each principal component of a voting system, enable a systematic development of trustworthy voting systems which can be provenly verified. We start this development with some selected formal requirements which are, e.g., translated from legal election regulations. The final yield of the considered development is a reliable voting software with formal guarantees that the given requirements are met, in a way that is comprehensible to informed users and domain experts. For the remaining part of this thesis, we focus on the scenario of a group of voters who, given their preferred choices, aim to elect one of multiple eligible alternatives. Albeit the intuitions and examples throughout this thesis primarily target the voting scenario, this is no principal or methodological limitation, and we occasionally indicate applications to other or more general use cases.

The Targeted View. The departing point of this thesis is the objective of developing a voting system in such a way that (1) a domain expert may afterward reliably comprehend that the requirements are met, (2) selections of the given requirements may be reliably justified or compared to other requirements, and (3) the trust obtained in objectives (1) and (2) may be reliably scrutinized by other domain experts.

The Underlying Component-Based View. Before explaining the contributions of this thesis, we need to substantiate our notion of a voting system. We refer to the component

model by Lundin (2010), where voting systems are "made up of parts from the four comparatively separable and distinguishable layers": (1) physical layer, (2) computational layer, (3) election layer, and (4) human layer. Each layer comes with different characteristics and expectations, and trustworthiness of a voting system cascades down the different layers. Lundin proposes to think about (electronic) voting systems as being component-based, and each layer further composes into separate components that each depend on a specific component in the layer below and realize a specific component in the layer above. The components by Lundin capture both the design and the implementation of a voting system, where at the basis hardware-based aspects are provided by the components in the physical layer, and finally at the top everything that faces the voter is realized in the human layer. In-between, the human expectations are derived, e.g., into legal regulations at the election layer, and are expected to be met by the software at the computational layer. Lundin furthermore assigns each component to a particular position in the system as a whole, where he distinguishes whether the component is close to and controlled by either the voter, which he denotes as *voter-close*, or the election authority, which he calls *authority-close*.

In the following, we focus on the step from the election layer to the computational layer and the guarantees that are expected from the regulations in the election layer to be met by the software at the computational layer. As our objective is to obtain a trustworthy voting system for comprehending, arguing, and scrutinizing in a reliable way, we want to make the involved software viable to such a process. The voting-specific problems are full of algorithmic and security challenges, and yet voting has a dramatic and broad impact on society, where the failure to address any of those challenges may undermine critical societal aspects, as trustworthiness of a voting system is paramount for the stakeholders to take part and accept the result.

It may still be feasible for the informed user or domain expert to comprehensibly scrutinize the legal regulations, but inspecting the software for the fulfillment of the regulations without being involved in its development may turn out to be infeasible and thereby unsatisfactory at large. This is exactly the trust gap that we aim to fill in this thesis. The involved components from the regulation, i.e., the election layer, are (a) the voter-ballot box communication channel that considers, e.g., how ballots are filled out, cast, confirmed, and transferred, (b) an election method, e.g., a simple rule where each voter chooses a single alternative and the one with the most votes wins, and (c) an election management system that includes the setup and instructions for all involved officials. Hence, we want the resulting cryptography schemes, tallying procedures, and anonymization strategies in the computational layer to reliably fulfill the requirements from the election layer.

1.2 State of the Art and Challenges

In the following, we present a short overview of the state of the art and associated challenges for each of those components both on the regulatory (election layer) and on the software (computational layer) side, as well as a brief description of the two remaining layers, i.e., the human layer and the physical layer. A more in-depth explanation and definition can be found in Chapter 3, with an overview of related work in Chapter 9.

1.2.1 Voter-Ballot Box Communication Channel

Description and Examples. The communication channel between voter and ballot box concerns how a voter casts a ballot. This component allows, e.g., for a high-level definition of the secrecy of the election, and describes the full procedure the voter goes through to cast a vote in a way that can be understood by the general public (Lundin, 2010). While by a secret election, we generally understand that the ballots[1]are secret (*ballot confidentiality*), we also want that the casting and processing of the ballots cannot be manipulated, i.e., we want to have assurance that the ballots are cast and processed exactly as the voters have filled them in (verified *ballot integrity*) (M. Bernhard et al., 2017). The two desiderata of assured ballot integrity and ballot secrecy are obviously conflicting with each other when taken to their full extent, as the more information we reveal about a voter's vote for being assured the vote is not manipulated, the more the secrecy of the vote is undermined (Koitmäe, Willemson, and Vinkel, 2021). On this account, various properties have been proposed in order to capture sufficient portions of the conflicting desiderata. For the sake of completeness, we note here that wide availability and usability of the channel between voter and ballot box are also commonly required, which are requirements that – following Lundin's component model – are to be dealt with in the physical and the human layer, respectively.

Specific security notions that capture aspects of ballot confidentiality are, e.g., coercion resistance, (everlasting) privacy, and receipt-freeness (M. Bernhard et al., 2017). Aspects of ballot integrity are captured by security notions such as ballot cast assurance, collection accountability, (verifiably) cast-as-intended, (verifiably) collected-as-cast, (verifiably) counted-as-collected, dispute-freeness, end-to-end verifiability (E2E-V), and software independence (M. Bernhard et al., 2017). Regarding verifiability, we note the distinction of individual and universal verifiability (Küsters and Müller, 2017). Individual verifiability denotes that the individual voter can determine whether their cast vote has correctly reached its destination, whereas universal verifiability denotes that the final election

[1]Even though secrecy of participation is also considered good practice, e.g., by the European Commission for Democracy through Law (Venice Commission) (2018, p. 22), since abstention is recognized to be a form of political choice by itself, concealing already the fact whether any given eligible voter actually participated is considered to be impossible to implement in practice (Koitmäe, Willemson, and Vinkel, 2021).

result correctly reflects the content of the (universally) collected votes. In order to properly address the above notions, we furthermore distinguish the four broad ballot casting categories of remote voting over the internet (also known as *i-voting*), in-place voting via electronic machines, postal remote voting via mail, and non-electronic in-place voting (e.g., in a voting booth). While the underlying challenge to balance integrity and confidentiality persists for all four categories, the individual categories have significantly different prerequisites and implications for each side.

As an illustration of the tradeoff between confidentiality and verified integrity of the ballot, take, e.g., in-place ballot casting. While filling out the ballot in the privacy of a voting booth may provide more secrecy of the individual ballot as eavesdropping attacks are significantly harder to implement at scale[1], mechanisms for individual voter verifiability are harder to implement as a ballot usually cannot be directly traced to the individual voter after the choice marked on the ballot has been added to the tally. As a means to mitigate this verifiability gap, one could use digital receipts. However, a receipt again increases the risk to the ballot's secrecy.

Cryptography Scheme. Voting procedures and the respective cryptography schemes that provide strong guarantees with respect to the above security notions of ballot integrity and confidentiality are an active area of research in cryptography. For analyzing and proving their security guarantees, cryptography schemecryptography schemes are usually modeled as one of two kinds of cryptographic games (D. Bernhard and Warinschi, 2014). A cryptographic game consists of a winning condition and an attacker, and a scheme is called secure with respect to the given notion of integrity or confidentiality if and only if– depending on the kind of game – one of the following holds: (1) no (active) attacker can win the game or (2) no (observing) attacker has more than random probability to guess the game's behavior The first kind of game is a *trace game*, in which the attacker wins if they do something that should be impossible in a secure system, e.g., obtain a voter's secret key or forge a signature on a message that was never actually signed by the signer. Moreover, the second kind of game is an *indistinguishability game*, in which the attacker is asked to guess in which of two options the game behaved, and they win if they guess the correct option with a probability of more than one half. For both kinds of games, the security notion of interest defines the resources the attacker has at hand, i.e., the moves that are available in the game, and the computational resources which the attacker can use to perform their moves. Such moves often involve well-known cryptographic primitives such as public-key encryption, homomorphic encryption, threshold techniques, digital signatures, zero-knowledge proofs, or verifiable shuffling. Regarding the attacker's computational resources, we often distinguish between unbounded attackers, who may

[1]Yet, even in-place casting can easily exhibit quite severe vulnerabilities that are easy to overlook, as has been demonstrated, e.g., by Ashur, Dunkelman, and Talmon (2016) for the Israeli general elections.

use unlimited resources, and polynomially bounded or efficient attackers, which is the more common class for practical voting systems.

M. Bernhard et al. (2017) provide an extensive overview over and evaluate various prominent voting systems and their employed cryptography schemecryptography schemes with respect to the above security notions. Therein, they cover both, poll-site voting procedures where voters record and cast ballots at predetermined locations, and remote voting procedures where voters fill out ballots anywhere and then send them (electronically or physically) to a central location to cast them. For poll-site procedures, these are, e.g., hand-counted in-person paper voting, direct-recording electronic voting (DREs), the ThreeBallot system (Rivest and W. Smith, 2007), and the system Prêt-à-voter (Ryan, Bismark, et al., 2009). For remote voting procedures, these are, e.g., the systems Helios (Adida, 2008), Civitas (Clarkson, Chong, and Myers, 2007), or Selene (Ryan, Rønne, and Iovino, 2016). A more in-depth explanation and definitions of the notions and schemes used within this thesis can be found in Chapter 3.

1.2.2 Election Method

Description and Examples. The election method concerns how the cast and collected ballots are interpreted and tallied, i.e., how the multiple individual choices by the voters are aggregated into a collective decision, which is the election result. First, this component includes the admissible choice by the voter, e.g., yes or no, one or multiple of a set of alternatives, or more complex forms such as an allocation of scores or a ranking of the alternatives. Second, we define admissible collective decisions, e.g., yes or no, a single winner, a ranking, or the election of a full parliament. Third and most important, this component includes a high-level description of the procedure that yields the election result from the individual choices as well as the respective requirements for this decision. Requirements may include simple properties such as parliament size, but may also contain more complex properties that express a desired intuition of fairness, such as proportionality of the result, capturing the will of the majority of the voters, or robustness against strategic voting. Procedures and desiderata for this component are an active area of research in the field of *social choice theory* that is at home in economic theory and is also analyzed in mathematics and algorithmic theory, which is situated in the field of *computational* social choice theory (COMSOC) (Brandt, Conitzer, et al., 2016; Endriss, 2017). Therein, such procedures are denoted as social choice functions or –more specifically– *voting rules*, and the desiderata are often captured in the form of formal axioms denoted as social choice properties.

The origins of social choice theory in economic theory lie in finding optimal decisions or distributions for a society and are also tied to mechanism design and game theory, where one is interested in stable market decisions, and the individuals may obtain

knowledge about the choices by other individuals and adapt their choices strategically with the goal of changing the collective decision, e.g., by the market, in their favor. For the matter of this thesis, we assume that the voters have stable individual preferences on all involved alternatives, generally in the form of a total linear order, and they do not know about the other voters' preferences. Yet, various social choice properties also capture forms of robustness against strategic voting, where certain non-truthful behavior is prohibited to lead to favorable outcomes. Notably, modern social choice theory is rooted in famous impossibility results, which state that strategic behavior cannot always be fully avoided (Gibbard, 1973; Satterthwaite, 1975) and already the attempt to obtain a small set of seemingly obviously desirable properties is theoretically impossible (Arrow, 1951). As a consequence, modern social choice theory developed a plethora of different voting rules that attempt to find good tradeoffs within this realm, and a variety of different social choice properties have been devised for this purpose.

Tallying Procedure. Prominent examples of tallying procedures in social choice theory range from the (simple) plurality rule, various Condorcet rules or the Borda rule, proportional procedures such as D'Hondt, Saint-Laguë or Single-Transferable Vote (STV), to algorithmically complex rules such as the Kemeny-Young rule. Practical adaptations and combinations of such rules often exhibit some form of undesirable behavior (Beckert, Goré, and Schürmann, 2013; Bundestagsdrucksache 17/11819, 2012).

1.2.3 Election Management System

Description and Examples. The election management system concerns how the voting system is set up, e.g., by some trusted officials, as well as control and communication of the whole process. This includes establishing the means and requirements that are needed by the other components, conducting an electoral register and the running alternatives, printing and distributing the ballot papers, securing and keeping up the election until it is finished, and finally announcing the result of the election. Common roles in the election management system are election provider, registrar, distribution facility, or election administrators, which are all typically designated by the election council.

Informally, we understand an election management system as everything that is done besides the actual voting. A trustworthy voting system entails that the means and requirements for the voting and tallying, i.e., the voter-ballot box communication channel and the election method, are also trusted by the voters and, e.g., for political elections, the running candidates. For simple elections with only a few voters who maybe all know each other personally or where the result is not critical, this might not be a concern at all, since the process can be easily understood and scrutinized by the voters themselves, or manipulation is simply not an issue. However, when elections have a larger scale or

the result may entail critical decisions, full oversight is not as easy or generally of high importance. In order to cope with that problem, the involved third parties need to be trusted and the election management system itself should provide means for scrutiny such that expected properties can be reliably checked, e.g., by the voters, for the voting system to be considered trustworthy.

Generally, we want this component to guarantee and preserve the desired properties described for the other components, and additionally that the selections of eligible voters, alternatives and maybe even of the third parties within election management are not compromised, e.g., by adding additional illegitimate ballots also known as *ballot stuffing*. Note that beyond guaranteeing that the third parties within the election management system do not manipulate the process, we may also be concerned that these parties gain unnecessary or any knowledge about confidential data within the voting system, e.g., the identities of the voters.

Unlike the other two – more fixed – components, our definition of the election management system component is not as clearly specified and does not directly translate into a single component within the computational layer. Within this thesis, we only consider *anonymization strategies* at this point, since they play an important role in electronic remote voting systems and can be defined in a relatively clear manner.

Anonymization Strategy. For ensuring prominent security properties in an electronic voting system, we must break the link between digital receipts and plaintext votes by performing an anonymization strategy. By these means, the voter's intentions are not revealed. Depending on the specifics, this component may heavily depend on the employed cryptography scheme for the communication between the voters and the ballot box. In some cases, they can also be defined independently, however. For example, when anonymization is done by a re-encryption mix network, the employed cryptography scheme can be chosen with some level of freedom (Lundin, 2010).

Another example for an anonymization strategy are decryption tables, which when combined with secret mixing ensures that the full link from an encrypted receipt to a plaintext vote is broken.

1.2.4 Human Layer and Physical Layer

Albeit they are not targeted within this thesis, yet for the sake of completeness, we now briefly describe the two remaining layers, namely the human layer and the physical layer.

Human Layer. The human layer concerns all aspects that are facing the (human) voter, e.g., the ballot-form configuration, management and layout of the polling station, or a verifiability front-end. This layer is of crucial importance in order to ensure both that all voters can participate and that the *usability* of the system's security procedures do not render it insecure due to the voters' failure to use them (Küsters and Müller, 2017). Moreover, the aspects facing the voters must also be simple and comprehensible in order to make sure that security and fairness – which may be obtained by formally proven guarantees – are also well-perceived and accepted by the voters.

Physical Layer. Finally, the physical layer is at the lowest level and supports all other layers with a physical infrastructure, e.g., a hardware authentication structure such as a public-key infrastructure with an asymmetric key pair, a publishing strategy, and a transfer method. Examples are cryptographic key pairs at the polling stations, the means to publish encrypted receipts by a central repository for them to be publicly accessible, or basic technical prerequisites as SSL encryption between polling stations and a central repository.

1.3 Contributions

This thesis contains five distinct contributions that are aligned according to the principal component of a voting system (as explained in Section 1.1) to which they belong, as illustrated in Figure 1.1.

The first component, the voter-ballot box communication channel, is addressed by contribution (I), which builds a bridge from the communication channel to the cryptography scheme by automatically generating secure card-based schemes from a small formal model with a parameterization of the desired security requirements.

The second component, the election method, is addressed by contributions (II) to (IV). Contribution (II) builds a bridge from the election method to the tallying procedure by automatically synthesizing a runnable tallying procedure from the desired requirements given as social choice properties. Contribution (III) is also right between the two layers to automatically generate either comprehensible arguments or bounded proofs to compare voting rules based on user-definable social choice properties and tallying procedures.

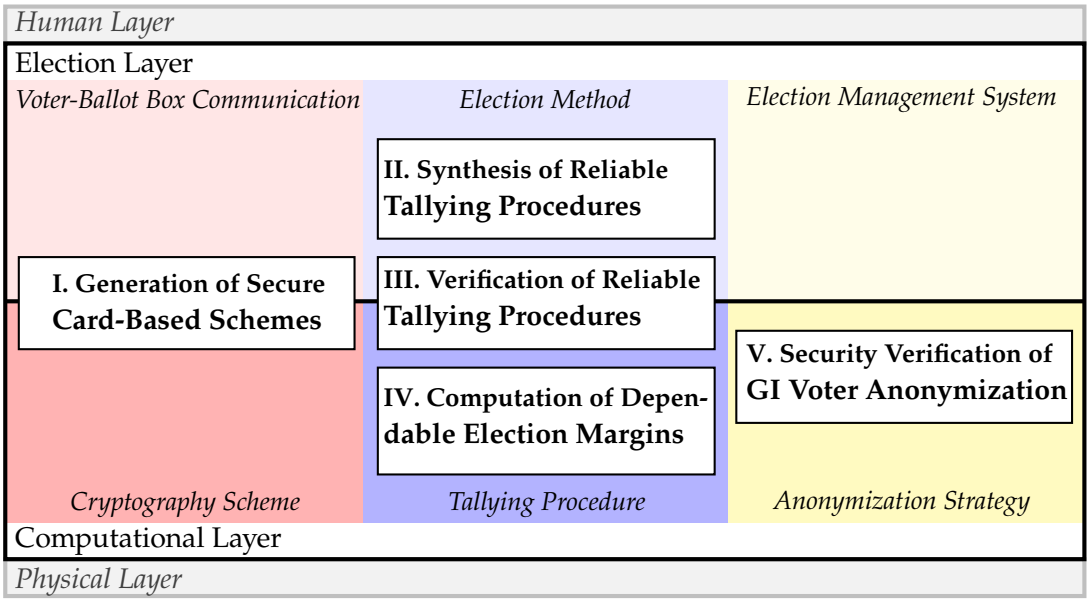


Figure 1.1: Contributions of this thesis

Moreover, contribution (IV) allows to automatically compute concrete election margins for a given tallying procedure, the collected ballots, and the computed election result.

Finally, the third and final component, the election management system, is addressed by contribution (V). Contribution (V) concerns the anonymization strategy and is a case study of a formal implementation-level security verification of the anonymization part of a real-world election management software employed for the annual elections of the German Informatics Society (GI – "Gesellschaft für Informatik").

The distinct contributions are:

(I) a method for the generation of secure card-based communication schemes,

(II) a method for the synthesis of reliable tallying procedures,

(III) a method for the efficient verification of reliable tallying procedures,

(IV) a method for the computation of dependable election margins for reliable audits,

(V) a case study about the security verification of the GI voter-anonymization software.

The presented methods assist the systematic development and verification of provenly trustworthy voting systems across traditional layers, i.e., from the election layer to the computational layer. They all pursue the goal of making voting systems trustworthy by reliable and explainable formal requirements. Since voting systems are inherently rich in algorithmic and security challenges, reliable and explainable formal requirements are

useful to allow that the inevitable tradeoffs can be made in a trustworthy manner. This thesis provides contributions for all the three respective system components on illustrative examples, and the devised formal methods provide translations of comprehensible requirements to reliable implementations of the components.

In the following, we give a brief overview for each of them.

1.3.1 Voter-Ballot Box Communication Channel

Generation of Secure Card-Based Schemes. At the very core, secure elections boil down to voters who want to determine a trusted collective result based on their secret individual choices without letting the others know more about their choice than what can be directly inferred from the result, i.e., a multi-party computation that is zero-knowledge. Here, we consider the illustrative scenario of only two voters who perform multi-party computation of a yes/no-decision with only a deck of cards. As already in this simple scenario the combinatorial state space may become very large, traditional proofs in literature tend to be involved and potentially error-prone. This contribution consists of a formal method that allows to automatically generate secure *card-based cryptography schemes* for a given amount of cards and protocol length, and can prove minimality for both parameters. In this work, we devise strict possibilistic formal guarantees that imply traditional cryptographic properties within an *honest-but-curious* model. The method is based on the lightweight formal technique of software bounded model checking (SBMC) (Section 2.2).

1.3.2 Election Method

Synthesis of Reliable Tallying Procedures. As soon as more than two parties are involved, however, a trusted result is based on more than securely computing or "tallying" a correct collective result. Depending on, e.g., legislative requirements, even devising the mathematical *social choice* function that determines the tally result from the individual choices may turn out to be non-trivial and error-prone. In this work, we devise simple trusted components and formal rules for the composition of such tallying procedures, such that desired requirements of the composed procedure are provenly guaranteed by the composition rules from properties of the underlying components. Our contribution is a formal method that automatically produces an executable tallying procedure from desired requirements and a computer-checkable proof that the procedure satisfies the requirements. The method is based on the formal techniques of interactive theorem proving (Section 2.4) and logic programming (Section 2.1).

Verification of Reliable Tallying Procedures. Yet, for scenarios where the desired requirements are not clear beforehand or need to be re-evaluated, we want to devise and optimize a social choice function based on potentially desirable requirements are of particular interest. This interest may either stem from particularly desirable or paradoxical behavior, and is demonstrated by particularly advantageous or disadvantageous voting situations. In this work, we devise a lightweight formal method that allows to dynamically adjust both the analyzed procedure and the requirement of interest, and generates comprehensible counterexamples whenever a requirement is not met within a given scope. Our contribution includes a simple way to express the requirements of interest and can be used by non-expert users to argue for or against tallying procedures based on comprehensible evidence in the form of counterexamples. The method is based on the lightweight formal technique of software bounded model checking (SBMC) (Section 2.2).

Computation of Dependable Election Margins. Elections do, however, involve multiple stakeholders, e.g., the running candidates, and a trustworthy voting system should be able to provide them with –ideally software-independent– evidence to gain trust in a fair and correct procedure and result. Such evidence can then be used under a transparent process of scrutiny or trial, so that trust in the result can be either comprehensibly established or comprehensibly refuted. In this work, we devise a general formal method that automatically computes election margins for a given election result and tallying procedure, which may then be applied in a trusted risk-limiting auditing process, e.g., by scrutinizing a limited amount of ballot papers. The method is based on the lightweight formal technique of software bounded modelchecking (SBMC) (Section 2.2).

1.3.3 Election Management System

Security Verification of GI Voter Anonymization. Further stakeholders, besides, e.g., the running candidates, are the voters. For simple elections, they may be able to observe and comprehend the whole process, as in the card-based example above. Yet, for more involved elections, eligible voters are usually identified by individual trusted credentials, which serve both for anonymously casting their ballot, and for gaining trust in or being able to challenge the collective result. In this work, we employ a formal method on software level to verify the security of voter-credential anonymization within an electronic voting system that has been used in the annual elections of the German Informatics Society (GI). Our contribution is a case study with the precise verification of secure information flow in an e-voting software, by which we evaluate applicability and scalability of formal methods with high precision for a real-world voting system. The case study is performed with the formal technique of deductive program verification (Section 2.3).

1.4 Previously Published Material

Significant parts, mainly Chapters 2 to 9, of this thesis have been previously published, either completely or partially.

1.4.1 Journal, Conference, and Workshop Publications

The content of Chapter 4 (“Generation of Secure Card-Based Communication Schemes”) has been first published by Koch, Schrempp, and Kirsten (2019) and further developed for more general decks and the verification of shuffle set size maximality by Koch, Schrempp, and Kirsten (2021), wherein the parts described in the chapter, i.e., regarding formal methods and formal verification, have been written mainly by the author of this thesis. Chapter 5 (“Synthesis of Reliable Tallying Procedures”) builds primarily upon the publication by Diekhoff, Kirsten, and Krämer (2020) with a short version by Diekhoff, Kirsten, and Krämer (2019), both written mainly by the thesis author, and has furthermore been extended by work developed jointly with Fabian Richter during and based on his master’s thesis (Richter, 2021). Moreover, the content of Chapter 6 (“Efficient Verification of Reliable Tallying Procedures”) builds upon the work by Beckert, Bormer, Kirsten, et al. (2016), and has been summarized by Beckert, Bormer, Goré, et al. (2017), both written mainly by the thesis author. Further examples and advances in Chapter 6 have been published by Kirsten and Cailloux (2018), which has also been written mainly by the thesis author. The Chapter 7 (“Computation of Dependable Election Margins for Reliable Audits”) has previously been published by Beckert, Kirsten, Klebanov, et al. (2017) and again summarized by Beckert, Bormer, Goré, et al. (2017), all of which has been written

in large parts by the thesis author. Finally, Chapter 8 ("Security Verification of the GI Voter-Anonymization Software") is original work by the thesis author that has been mainly unpublished, yet the setting and intentions have been published in a position paper by Beckert, Brelle, et al. (2019) and the results were presented at the 2019 annual meeting by the GI working group on formal methods and software engineering for secure systems (FoMSESS). Accordingly, the collection of previous works and foundations in Chapters 2 and 3 ("Formal Methods and Techniques" and "Preliminary Notions and Procedures"), as well as the collection of related works in Chapter 9 ("Related Work"), have for the most part been reworked from the appropriate parts –written mainly by the thesis author– in the publications mentioned above.

1.4.2 Software and Formal Proofs

The software for Chapter 4 has been written mainly by the thesis author. For Chapter 5, the formal proofs have been written mainly by Karsten Diekhoff, Jonas Krämer, Stephan Bohr and the thesis author, and the software thereupon by Fabian Richter (Richter, 2021), all of which has been planned and closely guided by the author of this thesis. The software for Chapter 6 has been planned and closely guided by the author of this thesis, and written mainly by Lukas Stapelbroek and Holger Klein. For Chapter 7, the software has been written mainly by the thesis author. Finally, the analyzed software in Chapter 8 has been provided by courtesy of Tomasz Truderung from the POLYAS company and translated meticulously from Kotlin to Java code by the author of this thesis. The formal proofs described in Chapter 8 have been conducted mainly by Florian Lanzinger and closely guided by the author of this thesis. The mentioned publications are cited where used and are also, for a convenient orientation to the reader, listed below.

1.4.3 Publications

Beckert, Bernhard, Thorsten Bormer, Rajeev Goré, Michael Kirsten, and Carsten Schürmann (2017). "An Introduction to Voting Rule Verification." In: *Trends in Computational Social Choice*. Ed. by Ulle Endriss. .II: Techniques. AI Access. Chap. 14, pp. 269–287. URL: http://research.illc.uva.nl/COST-IC1205/Book/.

Beckert, Bernhard, Thorsten Bormer, Michael Kirsten, Till Neuber, and Mattias Ulbrich (2016). "Automated Verification for Functional and Relational Properties of Voting Rules." In: *Sixth International Workshop on Computational Social Choice (COMSOC 2016)* (Toulouse, France, June 22–24, 2016). Ed. by Umberto Grandi and Jeffrey S. Rosenschein. URL: https://irit.fr/COMSOC-2016/proceedings/BeckertEtAlCOMSOC2016.pdf.

Beckert, Bernhard, Achim Brelle, Rüdiger Grimm, Nicolas Huber, Michael Kirsten, Ralf Küsters, Jörn Müller-Quade, Maximilian Noppel, Kai Reinhard, Jonas Schwab, Rebecca Schwerdt, Tomasz Truderung, Melanie Volkamer, and Cornelia Winter (2019). "GI Elections with POLYAS: a Road to End-to-End Verifiable Elections." In: *Fourth International Joint Conference on Electronic Voting (E-Vote-ID 2019)* (Lochau / Bregenz, Austria, Oct. 1–4, 2019). Ed. by Robert Krimmer, Melanie Volkamer, Bernhard Beckert, Véronique Cortier, Ardita Driza-Maurer, David Duenas-Cid, Jörg Helbach, Reto Koenig, Iuliia Krivonosova, Ralf Küsters, Peter Rønne, Uwe Serdült, and Oliver Spycher. Proceedings E-Vote-ID 2019. TalTech Press, pp. 293–294. URL: https://digi.lib.ttu.ee/i/?13563.

Beckert, Bernhard, Michael Kirsten, Vladimir Klebanov, and Carsten Schürmann (2017). "Automatic Margin Computation for Risk-Limiting Audits." In: *First International Joint Conference on Electronic Voting – formerly known as EVOTE and VoteID (E-Vote-ID 2016)* (Lochau / Bregenz, Austria, Oct. 18–21, 2017). Ed. by Robert Krimmer, Melanie Volkamer, Jordi Barrat, Josh Benaloh, Nicole J. Goodman, Peter Y. A. Ryan, and Vanessa Teague. Vol. 10141. Lecture Notes in Computer Science. Springer, pp. 18–35. DOI: 10.1007/978-3-319-52240-1_2.

Diekhoff, Karsten, Michael Kirsten, and Jonas Krämer (2019). "Formal Property-Oriented Design of Voting Rules Using Composable Modules." In: *6th International Conference on Algorithmic Decision Theory (ADT 2019)* (Durham, NC, USA, Oct. 10–27, 2019). Ed. by Saša Pekeč and Kristen Brent Venable. Vol. 11834.Short Papers. Lecture Notes in Artificial Intelligence. Springer, pp. 164–166. DOI: 10.1007/978-3-030-31489-7.

– (2020). "Verified Construction of Fair Voting Rules." In: *29th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2019), Revised Selected Papers* (Porto, Portugal, Oct. 8–10, 2019). Ed. by Maurizio Gabbrielli. Vol. 12042. Lecture Notes in Computer Science. Springer, pp. 90–104. DOI: 10.1007/978-3-030-45260-5_6.

Kirsten, Michael and Olivier Cailloux (2018). "Towards automatic argumentation about voting rules." In: *4ème Conférence Nationale sur les Applications Pratiques de l'Intelligence Artificielle (APIA 2018)* (Nancy, France, July 2–6, 2018). Ed. by Sandra Bringay and Juliette Mattioli. URL: https://hal.archives-ouvertes.fr/hal-01830911.

Koch, Alexander, Michael Schrempp, and Michael Kirsten (2019). "Card-Based Cryptography Meets Formal Verification." In: *25th International Conference on the Theory*

and Application of Cryptology and Information Security (ASIACRYPT 2019) (Kobe, Japan, Dec. 8–12, 2019). Ed. by Steven D. Galbraith and Shiho Moriai. Vol. 11921.I. Lecture Notes in Computer Science. Springer, pp. 488–517. DOI: 10.1007/978-3-030-34578-5_18.

Koch, Alexander, Michael Schrempp, and Michael Kirsten (2021). "Card-Based Cryptography Meets Formal Verification." Trans. by Takaaki Mizuki. *New Generation Computing* 39.1: *Special Issue on Card-Based Cryptography*, pp. 115–158. DOI: 10.1007/s00354-020-00120-0.

1.5 Structure of this Thesis

In the following, we give a brief outline for the rest of this thesis.

1.5.1 Part I—Introduction and Foundations

Within Part I, we introduce the terms and notions that are needed to understand the rest of this thesis in Chapters 2 and 3.

We start in Chapter 2 with the formal methods and according tools that are employed within this thesis and form the foundation for the targeted formal methods built therein. The foundations comprise logic programming in Section 2.1, software bounded model checking (SBMC) in Section 2.2, deductive!program verification of secure information flow in Section 2.3, and general proof assistants with human-readable proof artifacts in Section 2.4.

Within Chapter 3, we introduce the preliminary notions and procedures that are considered within this thesis, namely secure multi-party computation in Section 3.1, social choice theory in Section 3.2, risk-limiting audits and election margins in Section 3.3, and notions for end-to-end verifiability (E2E-V) in Section 3.4.

1.5.2 Part II—Secure Voter-Ballot Box Communication Channels

Part II introduces the first contribution on generating secure card-based protocols in Chapter 4.

We give the general notions and definitions in Section 4.1 and our employed computational model to express the security properties of interest in Section 4.2. Section 4.3 then explains the core methodology and the standardized model of our approach, with an evaluation in Section 4.4, the verification of bounds in shuffle set size to reduce complexity in Section 4.5, and summarize in Section 4.6.

1.5.3 Part III—Reliable Election Methods

Part III then considers election methods and tallying procedures, with our contribution to the synthesis of reliable tallying procedures in Chapter 5.

Section 5.1 introduces the individual elements for composing voting rules, before Section 5.2 describes the implementation in Isabelle/HOL. In Section 5.3, we describe the further steps based on logic programming for an automated synthesis tool. Finally, we evaluate our framework in Section 5.4 and summarize in Section 5.5.

In Chapter 6, we present our contribution for an efficient verification of tallying procedures with automatic counterexample generation.

Section 6.1 gives definitions and classifications of the desired properties, which we instrument in Section 6.2 for more efficient results. In Section 6.3, we show efficient techniques for relational verification, which we evaluate in Section 6.4. Finally, we show how to use these techniques for an argumentation approach based on counterexample generation in Section 6.5 and summarize in Section 6.9.

Moreover, Chapter 7 gives a contribution for automatic computation of election margins. Section 7.1 describes the approach, which is made more efficient in Section 7.3, with a larger case study in Section 7.4, which we summarize in Section 7.5.

1.5.4 Part IV—Secure Election Management Systems

Part IV finally considers the election management system with a case study on a real-world e-voting system in Chapter 8.

We describe the guarantees promised by the system in Section 8.1, illustrate how the system works in Section 8.2, and describe our specification and verification process in Section 8.3, which we summarize in Section 8.4.

1.5.5 Part V—Related Work and Conclusion

Part V finally concludes this thesis and discusses related work.

We give related work in Chapter 9, for formal methods regarding cryptographic protocols in Section 9.1, modular verification and synthesis of voting rules in Section 9.2, related work on the techniques for efficient verification and counterexample generation in Section 9.3, computation of election margins in Section 9.4, and verification of secure information flow on implementation level in Section 9.5.

We finally conclude in Chapter 10, with a summary in Section 10.1 and potential future work in Section 10.2.

1.5.6 Appendix

In Chapter A, we print card protocols found and analyzed within Chapter 4. Finally, Chapter B illustrates the structure of our Isabelle framework from Chapter 5.

2 Formal Methods and Techniques

> Le savant [ou la savante] doit ordonner ; on fait la science avec des faits comme une maison avec des pierres ; mais une accumulation de faits n'est pas plus une science qu'un tas de pierres n'est une maison.
>
> Henri Poincaré, *La Science et l'Hypothèse*, 1917

This chapter introduces the formal methods and techniques that are employed within the contributions of this thesis. We explain them starting with more lightweight up to more heavyweight formal methods, i.e., from most simple and automated, but limited in expressiveness, to more complex and interactive, but with a lot of expressiveness. For each of the methods, we illustrate and explain the specific techniques and tools which are used within this thesis.

2.1 Logic Programming

Logic programming is a formal technique that applies linear resolution and unification algorithms to answer a given query, based on a number of logical clauses, i.e., facts and rules (Apt, 1997). The algorithms are then used to derive a computation procedure that answers, i.e., *resolves*, the given query by successively applying rules to facts with the goal of generating the empty query based on the given one. When searching for a successful derivation of the query, the constructed successions of derivations – the algorithms' *search space* – form a derivation tree. The tree is successful if one of the tree's branches contains

the empty query (that cannot be derived any further), a – finite – tree is not successful if all its branches are failed derivations, i.e., they do *not* contain the empty query.

For the above technique to work efficiently and automatically, the logical clauses need to be of a certain "simple" form, namely *Horn clauses*, i.e., a disjunction of literals where at most one of them is not negated and all contained variables in a clause are universally quantified. Horn clauses can be written as simple logical implications, where a conjunction of nonnegated literals implies another nonnegated literal. Moreover, the query translates to a goal clause that contains no nonnegated literals, and each fact consists of exactly one nonnegated literal. These restrictions make the derivation problem of logic programming decidable so that no user interaction is necessary. Note that one caveat of the technique's efficiency is the *closed world assumption*, i.e., only queries that the rules can actually explicitly derive from the given facts are considered to be true.

Logic programming as described above is a simple form of theorem proving, but has also been established as a programming paradigm with *Prolog* being the most popular language where this paradigm is implemented.

The Prolog Programming Language

Prolog, which stands for "Programming in Logic," is a declarative programming language that implements the paradigm of logic programming based on linear resolution (Colmerauer and Roussel, 1993). While its prime target used to be natural language processing, it is nowadays used for many applications, for example in computational logic, and it is implemented in multiple standards and compilers. The language comprises a backtracking mechanism when retracting from one branch and searching the next one, as well as the possibility for a user to implement specialized control facilities, such as for defining their own search process as an alternative to the default *depth-first* search. Such specialized control facilities are implemented by so-called *meta-interpreters* that allow to evaluate Prolog's internal computation procedure with access to various additional intermediate steps, states, and variables. These additional capabilities make Prolog flexible and adaptable to numerous search and constraint problems, as long as the knowledge base can be expressed in the form of Horn clauses. With this flexibility, some Prolog implementations provide interfaces between Prolog and other programming languages. Thereby, e.g., Prolog's knowledge base can be precomputed from other contexts, or Prolog's computations can be employed within a greater software project that builds the computation into more complex services or libraries.

2.2 Software Bounded Model Checking

Software bounded model checking (SBMC) is a formal (program) verification technique that, given a program and a software property to be checked, verifies fully automatically whether the program satisfies the property (Biere and Kröning, 2018). In a nutshell, that question is translated into a reachability problem with respect to the given program.

SBMC symbolically, i.e., without the need for concrete values, executes the program and exhaustively checks it for errors that could violate the given property within some given bounds that restrict the amount of loop iterations and recursive method calls. Using these bounds, SBMC limits all runs through the program to a bounded length and can thereby unroll the control flow graph of the program and transform it into static single assignment (SSA) form (Clarke, Kroening, and Yorav, 2003). This bounded program is then translated into a formula in a decidable logic, e.g., an instance of the SAT problem. The formula is satisfiable if and only if a program run exists that violates the given software property within the given bounds.

Modern SAT or SMT solvers (Barrett and Tinelli, 2018; Gomes et al., 2008) can be used to check whether such a program run exists, in which case the SBMC tool constructs the corresponding problematic input and presents the counterexample to the user. If no such program run is found, that may be either because the property is actually satisfied, or because it is invalid only for runs exceeding the given bounds. In some cases, SBMC is also able to infer statically which bounds are sufficient, in order to come to a definitive conclusion.

SBMC tools also permit to extend the program with nondeterministic value assignments and `assume` statements in order to restrict the values and states that are to be considered. The properties to be checked are given in the form of `assert` statements. Hence, SBMC checks whether there are any runs through the program that satisfy all encountered `assume` statements but violate an `assert` statement. Let us now explain how SBMC can be instrumented for checking software properties.

Checking Software Properties

We assume we are given a procedure P under the form of an imperative program (for example, written in the C language), that uses some parameter values taken among a set of possible values I. An entry $i \in I$ is a list of values, one value for each such parameter: it gives a value to everything that a run of P depends on, such as its input variables, or anything that is considered nondeterministic from the point of view of P. For this reason, those parameters are qualified as *nondeterministic*, to distinguish them from normal parameters used in a programming language to pass information around. By contrast,

some values can be *derived*, thus, computed in P from the nondeterministic parameter values, or declared as constants in P, and both values of nondeterministic parameters or derived values can then be used as normal parameters in the program.

We are also given a software property to be checked about P, in the form $C^{\text{ant}} \Rightarrow C^{\text{cons}}$, where *ant* and *cons* stand for antecedent and consequence, respectively. Both C^{ant} and C^{cons} are sets of Boolean statements. A Boolean statement is a statement of P that evaluates to a Boolean value, for example, a statement checking that some computed intermediate value is odd. An entry i is said to satisfy a set of Boolean statements if and only if all Boolean statements in the set evaluate to true during the execution of P using the nondeterministic parameter values i, and is said to fail the set of Boolean statements otherwise. The property $C^{\text{ant}} \Rightarrow C^{\text{cons}}$ requires that for all possible entries $i \in I$, *if* i satisfies C^{ant}, *then* i satisfies C^{cons}. As an example, assume P computes, given i, two intermediate integer values v_1 and v_2, and then returns a third value v_3. The property to be checked could be: *if* v_1 is negative, *then* v_2 is positive and v_3 is odd. A solver that is asked to check a software property $C^{\text{ant}} \Rightarrow C^{\text{cons}}$ thus exhaustively searches for an entry i that satisfies C^{ant} but fails C^{cons}. The property is valid *iff* it is impossible to find such an entry.

2.3 Deductive Program Verification

Deductive program verification is based on a logical (program) calculus to construct a proof for a formula expressing that a program satisfies its specification (Filliâtre, 2011; Shankar, 2009). Typically, deductive program verification uses invariants and induction to handle loops. In order to mitigate complexity, most deductive approaches employ design by contract (Meyer, 1992), where functions respectively methods are specified with formal pre- and postconditions. These additional annotations enable a modular verification (Beckert, Bormer, Merz, et al., 2012), where each method is individually proved to satisfy its contract. To this end, each method – together with its contract – is translated into a formula, e.g., using some form of weakest precondition computation (Dijkstra, 1975). Method calls are replaced by the contract of the called method (instead of the method body), and loops are replaced by their invariants (instead of loop unwinding). The resulting formulae are either discharged using automated theorem provers, e.g., SMT solvers (Barrett and Tinelli, 2018), or shown to the user for interactive proof construction.

For the matter of this thesis, we are interested in using deductive program verification for the verification of secure information flow, which we express by the property of noninterference. In the following, we briefly introduce the noninterference property and

then give insights into how deductive verification can be used to verify the property on program level.

Noninterference. An established property guaranteeing confidentiality on code level is *noninterference*. Noninterference holds if no information flow from a secret input (of *high* security) to a public output (of *low* security) of the system is possible, i.e., if and only if no secret input of a program may influence its public output. Research on secure information flow dates back to the works by D. Denning (1976) and D. Denning and P. Denning (1977) and later Goguen and Meseguer (1982). We distinguish high variables containing secret data, which should be protected, from low variables, which are publicly readable, and introduce the *low-equivalence* relation ($\sim_L$) to characterize program states that are indistinguishable for any potential attacker. A program state s is an assignment of values to program variables and program locations. We assume that the input of a program is included in the program's initial state and that the output of a program is included in its final state. Two states, s and s', are low-equivalent if and only if all low variables in s have the same value as in s'.

Definition 2.1 (Noninterference) *A program P is noninterferent if and only if, for any initial states s_1 and s_2, the statement*

$$s_1 \sim_L s_2 \Rightarrow s_1' \sim_L s_2'$$

holds, where s_1' and s_2' are final program states after executing P in the initial states s_1 and s_2, respectively.

This means that two program executions starting in two low-equivalent states must terminate in two low-equivalent states, which guarantees that low outputs are not influenced by high inputs. Note that we restrict ourselves to terminating programs. In the following, we refer to noninterference with respect to a given high input and a given low output simply as noninterference.

2.3.1 Verification of Program Noninterference

Logic-based program analysis of information flow takes the semantics of the program language into account. The semantics of modern program languages provide a high degree of expressiveness, which must be considered when sources of illegal information leaks may be exploiting features of the program semantics. Logic provides a means for abstraction and can capture such features and moreover, using logical calculi, enables reasoning about their – direct or indirect, explicit or implicit – effects on any low program variables or locations. However, this requires a logical representation of the program

together with the precise property we want to prove. Using dynamic logic (Darvas, Hähnle, and Sands, 2005) together with symbolic values, we can express the functional property of *partial correctness* of a program P for a precondition ϕ and a postcondition ψ by the following formula:

$$\phi \rightarrow \left[P\right]\psi$$

This means that ψ holds in all possible states in which P terminates. Since we analyze only deterministic programs, this means that either P terminates and ψ holds afterward, or the program never terminates. Since we restrict ourselves to terminating programs, we only need to prove partial correctness in the following. Applying a logical calculus with a deductive theorem prover, we can hence symbolically execute P and attempt to prove the formula.

On this basis, we state the noninterference property based on value independence for a high variable h, a low variable l and a program P in the following way:

Definition 2.2 (Noninterference as value independence) *When $\mathtt{P}$ is started with values $\mathtt{l}$ that are arbitrary, then the value r of $\mathtt{l}$ – after executing $\mathtt{P}$ – is independent of the choice of $\mathtt{h}$ (note the order of the quantifiers).*

$$\forall \mathtt{l}\ \exists r\ \forall \mathtt{h}\ \left[\mathtt{P}\right]\ r = \mathtt{l}$$

However, instantiating existential quantifiers hinders automation and requires user interaction. As a mitigation, Barthe, D'Argenio, and Rezk (2004) established a noninterference formalization based on self-composition, effectively reducing it to a safety property. Using self-composition, the noninterference property of a program P translates to a safety property of a new program, which consists of P composed with a renaming of P.

Furthermore, we need to introduce the concept of state updates (Ahrendt et al., 2016), which capture the effects of symbolically executing program statements. We denote updates by variable assignments enclosed by curly braces, which are applied to logical terms and formulae, and thus change the program state.

We can now, based on the low-equivalence in Definition 2.1, extend our formalization of noninterference in Definition 2.3.

Definition 2.3 (Noninterference as self-composition with state updates)

$$\begin{aligned}\forall in_l\ \forall in_h^1\ \forall in_h^2\ \forall out_l^1\ \forall out_l^2\ \{l := in_l\}(&\\ \{h := in_h^1\}[P]\, out_l^1 = l&\\ \wedge\ \{h := in_h^2\}[P]\, out_l^2 = l&\\ \rightarrow\ out_l^1 = out_l^2)&\end{aligned}$$

Therein, we have two executions of P, one where the (high) program variable h is renamed to in_h^1, and another one where it is renamed to in_h^2. The (low) output variable l is captured in the variable out_l^1 after the first execution and in the variable out_l^2 after the second one. Finally, we need to prove that both outputs out_l^1 and out_l^2 are equivalent in the final state and assume equivalent low inputs via the variable in_l. The self-composition formula can hence be enclosed with purely universal quantifiers over the renaming variables for input and output. When trying to prove noninterference for a program P, theorem provers can now Skolemize these variables and greatly reduce the necessary user interaction.

Now, when dealing with object orientation, it is sometimes too strict to require all (low) variables and locations in the final state to be equivalent. For this matter, Scheben and Schmitt (2011) developed a variation of noninterference using a different semantics of low-equivalence based on an object isomorphism as defined in Definition 2.4. Therein, for any two states s_1 and s_2, and two isomorphisms π_1 and π_2, $\pi_1(o) = \pi_2(o)$ holds if o is observable in both states s_1 and s_2.

Definition 2.4 (Low-equivalence with isomorphism) *We call any two program states s, s'* low-equivalent *if and only if they assign the same values to any low variables (where L denotes the set of all low variables in the state s).*

$$s \simeq_L^{\pi} s' \quad \Leftrightarrow \quad \forall v \in L\ (\pi(v^s) = v^{s'})$$

The techniques described above together with this semantics are defined and implemented in the deductive program verification tool KeY for Java programs, which we explain in the following.

2.3.2 The KeY Verification System

KeY (Ahrendt et al., 2016) is a semi-interactive deductive theorem prover for statically verifying sequential Java programs formally specified using the Java Modeling Language (JML) (Leavens, Baker, and Ruby, 2006). In JML, each Java method can be annotated with pre- and postconditions, together with the program locations which the method's execution may change, such that verification can be done modularly. A successful verification using KeY proves that the method, when started in a state satisfying the precondition, terminates in a state satisfying the postcondition, and that it only affects those memory locations for which changes are allowed by the specification. Modularity is done at method level, i.e., method calls can be handled in proofs by replacing them with the logical translation of their JML contracts. Specification in JML allows using side-effect-free Java expressions in first-order predicates with extensions such as, e.g., quantifiers and abstract data types.

Verification is done by automatically translating the specified Java method into a – logically equivalent – formula in Java dynamic logic (JavaDL) (Beckert, Klebanov, and Weiß, 2016), where the source code can be executed and reasoned about inside the logic using symbolic execution to capture the method's possible effects in logical formulae.

KeY uses a sequent calculus, consisting of various deduction rules, which may either be applied automatically or interactively. As verification with KeY is done statically, i.e., without actually running the program code, a successful verification produces a formal proof which universally guarantees that, whenever the verified code is used in a setting where the specified precondition is satisfied, the program will satisfy the specified postcondition and change at most the specified memory locations. An advantage of static analysis is that no run-time overhead is incurred as is the case with other approaches such as runtime monitoring.

The KeY tool supports both automated and interactive verification, and proofs can often be constructed automatically by the built-in strategies based on configurable heuristics and various predefined macros. KeY's support for user interaction permits deductive program verification with respect to expressive specifications, e.g., for more involved proof obligations such as security properties. In general, the problem is undecidable and verification sometimes requires some share of the rule applications to be decided by user interaction. KeY is capable of verifying noninterference for Java programs and covers a wide range of Java features (Beckert, Bruns, et al., 2013). With this toolkit, powerful specification elements are given for proving noninterference, also allowing for declassification.

Efficient noninterference proofs are supported using modularization via the *design-by-contract* concept with an extension of the Java Modeling Language (JML). Such a contract specifies the low program variables and locations for the initial and the final state of the specified program part. The proof obligation hence requires the low elements in the final state to depend at most on the low elements in the initial state. When using the semantics for object isomorphisms, these contracts may also contain a list of fresh objects to be included in the isomorphism.

2.4 Interactive Theorem Proving

All the formal techniques introduced above fall into the category of computer-assisted theorem proving with some restrictions to either automate the task or reduce proof complexity by specialized formal languages, mainly to reason about computer programs. However, when we abstract away from computer programs, we require more expressiveness. As our formalisms and proof obligations become more expressive, this should also apply to the proofs themselves, including the possibility of considerable interactions on the proof itself that are similar to interactions when writing a program (Paulson, 1989).

For this purpose, we employ a general-purpose interactive theorem prover, often simply called a generic *proof assistant*. Supplementary to rigorous proof calculi, a generic proof assistant allows the user to express their own formal model as well as targeted deductive rules and theorems on this model. With this high level of freedom, it becomes even more important that the mechanism for checking proofs is rigorous and trusted, and provides clear feedback to the user as well as flexible methods of proof interaction. Moreover, proofs within generic proof assistants should provide a good level of readability for humans so that the user does not lose focus and can guide the proof efficiently and successfully.

The Isabelle/HOL Proof Assistant

The Isabelle/HOL system is an established generic proof assistant which provides a generic infrastructure for implementing deductive systems in higher-order logic (HOL) and enabling to write tactics for human-readable and machine-checked proofs to show that the deductive conclusions are indeed correct (Nipkow, Paulson, and Wenzel, 2002). Higher-order logic is a language for writing formal mathematics and allows defining very expressive, rigorous, and general theorems. Isabelle comes with a small trusted core of rules in HOL, as well as extensive machine-checked libraries of mathematical theories with flexible and strong assistance for writing proofs. Proofs are mostly written in the Isar language which allows addressing the deduction rules in a relatively natural (to a mathematician) way, combined with many flexible and rigorous tactics that enable reasoning with the theories. By this means, a theorem written in Isar is – once proven correct within Isabelle – re-checked and confirmed by Isabelle/HOL within a few seconds every time the theorem is loaded.

Moreover, Isabelle comprises many automated techniques, e.g., based on SMT solvers, that aim to make the proof search as natural as possible. Besides proof writing and checking, Isabelle also contains a functionality to translate mathematical functions into verified executable software programs which are proven to be correct with respect to the theories in Isabelle.

3

Preliminary Notions and Procedures

Asking software engineers about computerized voting:
#1—"That's *terrifying*."
"Wait, really?" #1—"Don't trust voting software and don't listen to anyone who tells you it's safe."
"Why?" #1—"I don't quite know how to put this, but our entire field is bad at what we do, and if you rely on us, everyone will die."
"They say they've fixed it with something called 'blockchain.' " #1—"Aaaaa!!!"
#2—"Whatever they sold you, don't touch it."
#1—"Bury it in the desert."
#2—"Wear gloves."

Randall Munroe, *xkcd #2030*, 2018

Before describing the main contributions of this thesis, i.e., how we instrument the formal methods in Chapter 2 to devise the targeted formal methods in this thesis, we need to first introduce some preliminary notions and procedures. In the following, we start by a brief description of secure multi-party computation in Section 3.1 for a better understanding of the contribution within Chapter 4. We then give the core concepts and definitions of social choice functions in Section 3.2 that are necessary to understand the three contributions within Part III. The third contribution in that part, i.e., the formal method for margin computation in Chapter 7, furthermore requires some understanding of risk-limiting audits and election margins, which we provide in Section 3.3. Finally, we briefly define security notions under the umbrella of end-to-end verifiability (E2E-V) as

well as software independence in Section 3.4, which are useful to understand the design of the e-voting system which we analyzed within the case study presented in Chapter 8, our fifth contribution.

The contents of Section 3.2 have been extracted from previously published works by Diekhoff, Kirsten, and Krämer (2020), Kirsten and Cailloux (2018), Beckert, Bormer, Goré, et al. (2017), and Beckert, Bormer, Kirsten, et al. (2016). Finally, the contents of Section 3.3 have been extracted from previously published works by Beckert, Kirsten, Klebanov, et al. (2017) and Beckert, Bormer, Goré, et al. (2017).

3.1 Secure Multi-Party Computation

As described earlier in Section 1.2, security of the communication between the voters and the ballot box is particularly interesting as it combines the need for multiple desirable requirements that are naturally in conflict with each other. One cryptographic technique which provides particularly strong security guarantees for this communication, given at least semi-trusted election authorities and no particular assumptions on the election method, is secure multi-party computation (W. Smith, 2004). Within multi-party computation that is based on zero-knowledge proof protocols, we assume a number of mutually distrustful parties that each provide a shared secret which is never known by any other individual party or collusive subset, which amounts at most to some given threshold number. Their goal is to compute a common forward-flow Boolean circuit of logic gates, that is publicly known, based on their (shared) secrets. Within the computation process, all the intermediate logical states and the output bits are shared secrets together with the individual secret input bits. Furthermore, the computation guarantees that any external observer, who views the parties communicating with each other, can become convinced that the computation and thus the final result is indeed correct, but does not learn anything non-trivial about any of the secret bits. Regarding complexity, the computational work as well as the total amount of communicated bits are polynomial with respect to the amount of parties and logic gates involved, and the individual work is preferably roughly the same for each party. Nonetheless, any subset of the parties that is larger than the given threshold number is easily able to cooperate for deducing and revealing any desired subset of the shared secrets, particularly the output bit (which is the common case).

We have described the general technique of secure multi-party computation, which provides both a correctness and a zero-knowledge guarantee. A noteworthy concretization that enables secure multi-party computation is a *mix network* (also called *mixnet*). Mix networks themselves do not perform the actual computation, but they hide the link between voter and ballot in such a way that subsequently, the votes can be decrypted, and

then counted to compute the election result on a bulletin board for public view and hence seen by everybody to be correct (W. Smith, 2004). The primary operation within mix networks is shuffling, originally performing a sequence of individual permutations on, the ballots so that the ballots' order is random, and every involved party can be confident that the ballots are shuffled, but no party knows the actual shuffle-permutation(s). For each shuffle-permutation, a zero-knowledge proof is produced as a guarantee that no ballots are exchanged, removed, or added on the way. The described network allows mixers to be mutually distrustful, as even if some parties may know some permutations, nobody can know the final product permutation. For large elections, mix networks must potentially perform a large amount of communication and computation, but they can handle arbitrary ballot formats (D. Bernhard and Warinschi, 2014). Nonetheless, as the actual vote counting is seen by everybody, secure mix networks require that individual votes are *anonymizable*, i.e., filling out a ballot in a *unique*[1] way is practically infeasible or useless.

Regarding the specifics, but also besides mix networks, there are many concretizations for the model of (secure) multi-party computation that may each differ in various ways. We will see one particularly noteworthy concretization using physical objects in Chapter 4. Moreover, in Chapter 8, we analyze an electronic voting system that uses a mix network.

3.2 Social Choice Functions

In the following, we summarize some core concepts and definitions from social choice theory for a more refined goal of the corresponding contributions of this thesis. For an in-depth overview, we refer the interested reader to two comprehensive books by Brandt, Conitzer, et al. (2016) and Endriss (2017), as well as the seminal paper by Arrow (1951).

3.2.1 Voting Rules

We consider a fixed finite set $\mathcal{A}$ of eligible *alternatives* and a finite (possibly ordered) set $\mathcal{K}$ of *voters* (with cardinality k). In an election, each voter i casts a *ballot* $\succsim_i \in \mathcal{L}(\mathcal{A})$, which is a linear order[2], ranking the alternatives $\mathcal{A}$ according to i's preferences. We collect all votes in a *(preference) profile*, i.e., a sequence $\succsim = (\succsim_1, \ldots, \succsim_k)$ of k ballots. Given the set $\mathcal{L}(\mathcal{A})$ of linear orders on $\mathcal{A}$, $\mathcal{L}(\mathcal{A})^k$ defines the set of all profiles on $\mathcal{A}$ of length k, i.e., for all voters. Hence, we have $\mathcal{L}(\mathcal{A})^+ = \bigcup_{k \in \mathbb{N}^+} \mathcal{L}(\mathcal{A})^k$, the set of all finite, nonempty profiles on $\mathcal{A}$, i.e., the input domain for a voting rule. Voting rules (see Definition 3.1) elect a

[1] The discovering of a voter identity by the pattern of their ballot is sometimes called the *Italian attack*, after a once prevalent practice in Sicily.

[2] A linear order is a transitive, complete, and antisymmetric relation.

nonempty subset $\mathcal{C}(\mathcal{A})$ of the alternatives as (possibly tied) winners, given $\mathcal{C}(X)$ denotes the set of all nonempty subsets[1] of a set X.

Definition 3.1 (Voting rule) *Given a finite set of alternatives $\mathcal{A}$, a* voting rule *f maps each possible profile $\succsim \in \mathcal{L}(\mathcal{A})^+$ to a nonempty set of winning alternatives in $\mathcal{C}(\mathcal{A})$:*

$$f : \mathcal{L}(\mathcal{A})^+ \to \mathcal{C}(\mathcal{A}).$$

In practice and in literature, a multitude of different voting rules are in use. A common example is the function that returns all alternatives that are ranked at first position by a plurality of the voters, hence called *plurality voting*. Another common kind of voting rules assigns values for every ballot to each alternative according to their position occupied on the ballot, and elects the alternatives with the maximal score, i.e., the sum of all such values for them. Such rules are called *scoring rules*, e.g., the *Borda rule*, where the value of an alternative on a ballot is the amount of alternatives ranked below them on that ballot.

From this, we can define the three classical voting rules by Borda, Black, and Copeland in the following.

Definition 3.2 (Borda rule) *The Borda rule, given a profile $(\succsim_i)_{i \in N}$, associates to each alternative a and voter i the score $s(a, i)$ equal to the amount of alternatives that a beats in $\succsim_i$, and associates to each alternative a the score $s'(a) = \sum_{i \in N} s(a, i)$. The winners are the alternatives that have the maximal score: $f_{Borda}((\succsim_i)_{i \in N}) = \arg\max_{a \in \mathcal{A}} s'(a)$.*

Definition 3.3 (Black's rule) *The Black (1958, p. 66) rule selects the Condorcet winner if there is one, otherwise, the Borda winners.*

Given a profile $R = (\succsim_i)_{i \in N}$, let $M_R(a)$ denote the set of alternatives against which a obtains a strict majority, and ${M_R}^{-1}(a)$ the set of alternatives that obtain a strict majority against a.

Definition 3.4 (Copeland rule) *The Copeland (1951) rule (actually a close variant of a rule proposed by Ramon Llull in the 13th century (Colomer, 2013)), given a profile R, gives to each alternative the score $s(a) = |M_R(a)| - |{M_R}^{-1}(a)|$, and lets the alternatives with maximal score win.*

[1] In contrast, the power set $\mathcal{P}(X)$ also includes the empty set, but is otherwise identical to $\mathcal{C}(X)$.

3.2.2 Social Choice Properties

Within social choice theory, the axiomatic method has established a number of general fairness and reliability properties called (*axiomatic*) *social choice properties*. They formally capture intuitively desirable or in other ways useful properties to compare, evaluate, or characterize voting rules. Such properties are applicable in a general way, as they are defined on abstract voting rules only with respect to profiles and returned sets of winning alternatives. For the sake of simplicity, the examples illustrated in the following only address properties of universal nature, i.e., they require that all mappings of a given voting rule belong to some set of admissible ways, as formally described by the property of interest, for associating sets of winners to profiles. Besides properties which *functionally* limit the possible sets of winning alternatives for any one given profile, properties may also *relationally* limit combinations (of finite arity) of mappings, e.g., certain (hypothetical) changes of a profile may only lead to certain changes of the winning alternatives. Relational properties capture a voter's considerations, such as how certain ways of (not) filling out their ballot may or may not affect the chances of winning for some alternatives.

In the following, we introduce *Condorcet consistency* and *monotonicity* as they serve as running examples later on. Note here that the three previously defined voting rules Borda, Black, and Copeland, all comply to Condorcet consistency. The functional property *Condorcet consistency* (see Definition 3.6) requires that if there is an alternative w that is the *Condorcet winner* (see Definition 3.5), the rule elects w as unique winner. A Condorcet winner is an alternative that wins every pairwise majority comparison against all other alternatives, i.e., for any other alternative, there is a majority of voters who rank the Condorcet winner higher than that alternative.

Definition 3.5 (Condorcet winner) *For a set of alternatives $\mathcal{A}$ and a profile $\succsim \in \mathcal{L}(\mathcal{A})^+$, an alternative $w \in \mathcal{A}$ is a* Condorcet winner *if and only if the following holds:*

$$\forall a \in \mathcal{A} \setminus \{w\} : |\{i \in \mathscr{K} : a \succsim_i w\}| < |\{i \in \mathscr{K} : w \succsim_i a\}|.$$

Note that, if a Condorcet winner exists, it is unique by the above definition.

Definition 3.6 (Condorcet consistency) *For a set of alternatives $\mathcal{A}$, a voting rule f is* Condorcet consistent *if and only if for every profile $\succsim \in \mathcal{L}(\mathcal{A})^+$ and (if existing) the respective Condorcet winner $w \in \mathcal{A}$, the following holds:*

$$w \textit{ is } \text{Condorcet winner } \textit{for } \succsim \quad \Rightarrow \quad f(\succsim) = \{w\}.$$

Note here that for profiles for which no Condorcet winner exists, the property imposes no requirements on the election outcome. The relational property *monotonicity* expresses that

if a voter were to change their vote in favor of some other alternative, the outcome could never change to the disadvantage of that alternative. Monotone voting rules are resistant to some forms of strategic manipulation, where a voter could make their preferred alternative the (unique) winner by misrepresenting their actual preferences and assigning a higher rank to another alternative on their ballot. A voting rule is monotone (see Definition 3.8) if and only if for any two profiles $\succsim$ and $\succsim'$ which are identical except for one alternative a that is ranked higher in $\succsim'$ (while preserving all remaining pairwise-relative rankings), the election of a for $\succsim$ always implies their election for $\succsim'$. We define this "ranking higher" as *lifting* an alternative (see Definition 3.7).

Definition 3.7 (Lifting) *For a set of alternatives $\mathcal{A}$ and two profiles $\succsim, \succsim' \in \mathcal{L}(A)^k$, $\succsim'$ is obtained from $\succsim$ by* lifting *an alternative $a \in \mathcal{A}$ if and only if there exists a ballot $i \in [1, k]$ such that $\succsim_i \neq \succsim'_i$ and for each such i the following holds:*

i. There exists some alternative $x \in \mathcal{A}$ such that $x \succsim_i a$ and $a \succsim'_i x$, and

ii. we have $y \succsim_i z \Leftrightarrow y \succsim'_i z$ for all other alternatives $y, z \in \mathcal{A} \setminus \{a\}$.

We may thus define the monotonicity property as follows.

Definition 3.8 (Monotonicity) *For a set of alternatives $\mathcal{A}$ and an alternative $a \in \mathcal{A}$, a voting rule f is* monotone *if and only if for all profiles $\succsim, \succsim' \in \mathcal{L}(\mathcal{A})^+$ where $\succsim'$ is obtained from lifting a in $\succsim$, the following implication holds:*

$$a \in f(\succsim) \Rightarrow a \in f(\succsim').$$

3.2.3 Seat Apportionment Methods

In the event of electing multiple representatives instead of a single winner (or a tie between some of them), e.g., for a parliament, the above terms must be adapted. Such procedures are commonly denoted as *seat apportionment methods*. Supplementary to the above desiderata, seat apportionment methods are commonly required to elect a sufficiently *proportional* selection of winners. For a comparison of different seat apportionment methods, we refer the interested reader to the work by Gallagher (1991), who compares various such methods on the basis of different kinds of measures to reflect their degrees of proportionality.

One example of a method for seat apportionment that is widely-used, e.g., on various political levels in Europe, is the D'Hondt or Saint-Laguë method (Stark and Teague, 2014). The D'Hondt method is a divisor method, i.e., tabulated votes are divided by a given *divisor* and rounded in a particular manner, and is applied after vote counting and

tabulation sorts the votes into stacks where each stack contains votes for a single political party. The input then is the amount of votes for each party, i.e., the amount of votes in the corresponding stack. The D'Hondt method proportionally allocates mandates to parties in such a way that the amount of votes represented by mandates is maximized, i.e., the votes-per-seats ratio – intuitively the price in amount of votes to be paid by a party to get one seat – is chosen as high as possible while still allocating all seats in parliament. By this means, D'Hondt achieves an – as far as possible – proportional representation in parliament (Gallagher, 1991).

D'Hondt can be implemented as a *highest-averages* method in the following way: The amount of votes for each party is divided successively by a series of divisors, which produces a table of quotients (or averages). In that table, there is a row for each divisor and a column for each party. For the D'Hondt method, these divisors are the natural numbers $1, 2, \ldots, m$, where m is the total amount of mandates to be distributed. Then, the greatest numbers in the quotient table – respectively the parties in whose columns these numbers are – are each allocated one seat. The "final" seat goes to the m'th greatest number. Hence, the threshold level of the votes-per-seats-ratio lies in the interval between the m'th greatest number and the $(m+1)$'st greatest number of all computed averages in the quotient table.

Alternatively, the D'Hondt method can also, equivalently, be described without a quotient table. Instead, a quota is chosen, i.e., an amount of votes needed to "buy" one mandate, such that the resulting mandates per party, when rounded down to the next natural number, sum up to the required total amount of mandates. This is known as Jefferson's method and is similar to *largest-remainder* methods such as the Hare-Niemeyer method. The quota corresponds to the lowest quotient in the D'Hondt table for which a mandate is allocated.

Yet, there are many more methods of seat apportionment. We refer the interested reader to Pukelsheim (2017) and Balinski and H. Peyton Young (2010) for a detailed overview.

3.3 Risk-Limiting Audits and Dependable Evidence

A risk-limiting audit is a statistical method to create confidence in the correctness of an election result by checking samples of paper ballots. Lindeman and Stark (2012) distinguish *ballot-polling audits*, where they draw a carefully chosen random sample of ballots to check whether the sample gives sufficiently strong evidence for the correctness of the published election result. In contrast, a *comparison audit* checks the ballot interpretation for a random sample during the audit against the ballot's respective interpretation in a vote-tabulation system.

Both auditing techniques, ballot-polling and comparison audits, rely on the availability of the ballot manifest which describes in detail how the ballots are organized and stored, including how many stacks there are and how many ballots can be found in each stack. This information is needed for drawing the sample. In addition, we need to know what the *election margin* is, i.e., the amount of votes that would need to be changed in order to change the election outcome. This is also the amount of votes that would have had to be miscounted or tampered with in order to change the election outcome. If the election margin is large, only a small ballot sample needs to be audited. If it is small, the required sample size increases.

We assume that the election function that we consider satisfies the anonymity property, i.e., identical ballots have the same effect on the election outcome. Then, for a given election with TOTAL votes, during the counting process, the votes are accumulated into stacks $S_1, \ldots, S_k$, where each stack holds p_i identical votes ($p_i \geq 0$ is the size of S_i) and $\texttt{TOTAL} = \sum_i p_i$. This allows us to use $\langle p_1, \ldots, p_k \rangle$ as input to the election function. In the following, we assume that each stack is associated with a political party and that PARTIES is the total amount of running parties, i.e., $k = \texttt{PARTIES}$ (there can also be stacks for special cases such as invalid votes, which we would treat equivalently to parties in the following). We call $\langle p_1, \ldots, p_k \rangle$ the vote table for the election. The election margin is the smallest amount of votes that need to be put on stacks different from where they are in order to change the outcome of the election.

Definition 3.9 (Election margin) *The* election margin *for an election function f and a vote table $\langle p_1, \ldots, p_k \rangle$ is the smallest number* MARGIN *such that there is a vote table $\langle p'_1, \ldots, p'_k \rangle$ with*

$$f(\langle p_1, \ldots, p_k \rangle) \neq f(\langle p'_1, \ldots, p'_k \rangle)$$

and

1. $\texttt{MARGIN} = \sum_{i=1}^{k} d_i$ *where* $d_i = p'_i - p_i$ *if* $p'_i > p_i$ *and* $d_i = 0$ *otherwise.*
2. $\sum_{i=1}^{k} p'_i - p_i = 0$

The first condition in the above definition ensures that the total amount of votes that are moved between stacks is of size MARGIN. Furthermore, the second condition ensures that votes are moved from one stack to another and are not created or removed.

Besides the (global) margin defined above, more specialized audits might require other margins than the ones defined in Definition 3.9, i.e., which are defined by different types of changes in the vote table or by particular effects on the election result. For example, one may compute the margin for increasing the amount of mandates allocated to a particular party.

Instead of distinguishing between different types, in the following we focus on *two-vote overstatements* of the margin, as these are suitable for a variety of election functions. An audited ballot is a *two-vote overstatement* if it witnesses simultaneously two mistakes, namely that it was counted wrongly towards someone who won, while it should have been counted towards someone who lost. In contrast, a *one-vote overstatement* refers to a ballot that was erroneously not counted towards the loser, but neither was it counted towards the winner. For the purposes of this thesis, both one-vote and two-vote overstatements are counted as one change in the vote tabulation. However, this can be extended to distinguish between the two types of error, but we want our approach to be general and the distinction between one-vote and two-vote overstatements does not exist for all election functions (e.g., approval voting).

Next, we review the statistics underlying margin-based risk-limiting audits following Stark (2010). Risk-limiting audits are performed in stages. At every stage, the theory requires that we audit at least $n = \rho/\mu$ ballots, which is also called the *sample size*. The value ρ is called the *sample-size multiplier* and defined below. Each ballot is randomly chosen among all the ballots, and the audit verifies that they were each counted for the correct stack S_i. The fraction μ refers to the *diluted margin*, i.e., the percentage of votes that would have to be changed to change the election outcome. It is computed as $\mu = \texttt{MARGIN}/\texttt{TOTAL}$, where MARGIN is the election margin (Definition 3.9), and TOTAL is the total amount of ballots cast.

Before the audit can start, a set of auditing parameters needs to be determined, which allows us to calculate the size of the sample to be drawn. The *auditing parameters* include

- the *risk limit* α, which determines the largest chance that an incorrect outcome will not be corrected by the audit (if we want to be 99% sure that the election outcome is correct, then we choose $\alpha = 0.01$);
- the *error inflation factor* γ, which controls the tradeoff between initial sample size and the additional counting required if the audit finds too many errors;

- and lastly the *tolerance factor* λ, which describes the tolerance towards errors; it is the amount of detected errors that is tolerated, expressed as a fraction of the election margin (i.e., $\lambda = 0.1$ means that 5 errors are tolerated when `MARGIN` = 50).

Finally, we have everything in place needed to define the sample-size multiplier ρ, which only needs to be computed once for each audit, as follows:

$$\rho = \frac{-\log \alpha}{\frac{1}{2\gamma} + \lambda \log(1 - \frac{1}{2\gamma})}.$$

In summary, the auditing process as described by Stark (2010) adheres to the following steps:

First, the auditor commits values for α, γ, and λ and computes the value ρ as shown above. Then, the diluted margin μ is computed, which explicitly depends on the election margin `MARGIN`. Next, the real audit commences by drawing the sample of size $n = \rho/\mu$ at random. If the audit encounters too many errors (more than $\lambda * \texttt{MARGIN}$), a new stage is triggered, with a sample size that is increased by the factor γ; otherwise the audit is successfully concluded. In the worst case, the technique proceeds to a full hand-count when the sample size exceeds `TOTAL`. For a more detailed description on how to compute how much the sample must grow from stage to stage, consult Stark (2010).

In all of this, the true challenge is to compute the correct election margin. Different election functions require different margin computations, and for many, an algorithm to compute the margin is unknown.

3.4 End-To-End Verifiability and Software Independence

While we have described in Section 3.1 how to do secure voting using shared secrets, this becomes quickly impractical for real-world elections with more than a handful of voters. With complex systems and many stakeholders, it also gets more demanding for the individuals, e.g., the voters, to convince themselves that the election is carried out as expected. For this matter, common security notions are concerned to provide convincing evidence as built-in functionality, e.g., the notion *end-to-end verifiability (E2E-V)* denotes that each individual voter can themselves monitor the integrity of the election (M. Bernhard et al., 2017). From one end to the other, this comprises that voters can independently verify (a) that their votes are correctly recorded (*cast as intended*), (b) that the representation of their vote is correctly collected in the tally (*collected as cast*), and (c) that every well-formed and collected vote is correctly included in the tally (*tallied as collected*).

E2E-V requires furthermore that it is possible to check the list of those voters who cast ballots, such that no *ballot-box stuffing* can occur, i.e., no additional votes are added to the

collection (*eligibility verifiability*). The first two monitoring mechanisms are also commonly classified as *individual verifiability* since an individual voter can verify their own vote, and the third one as *universal verifiability* since the collection of votes can be verified as a whole.

Oftentimes when talking about verifiability notions in voting systems, also the term *accountability* comes up, since it is based on verifiability mechanisms. For example, *collection accountability* denotes that, once a voter detects that their vote has not been collected as cast or intended within the vote!castingvote-casting protocol, they obtain evidence that is convincing to an independent party in order to demonstrate that their vote has not been correctly collected. Yet, when aiming to provide accountability, there is a likely tradeoff with the confidentiality requirement of non-coercibility or coercion resistance, since the voter might be forced to present the obtained evidence as a convincing argument to a (malicious) coercer.

Another popular requirement directly concerns the mechanisms that provide the guarantees of individual, universal, and eligibility verifiability, namely the notion of *software independence* (Rivest, 2008). While this term concerns not only monitoring, but also auditing mechanisms, it is of particular cryptographic interest for the various notions of verifiability. Software independence requires for a voting system that undetected changes or errors in its software cannot lead to undetectable changes or errors in the election outcome. Once this further requirement is met, the monitoring mechanisms become fully reliable without the need to inspect the software. However, it can also be argued that this requirement only shifts the blame from the voting system to the monitoring mechanisms, which are commonly also implemented as software. An even stronger form of software independence is the notion of *strong software independence*, wherein any detected change or error in the election outcome that is caused by the software can be corrected without rerunning the election. Hence, if this requirement is met additionally to (regular) software , the voting software has a means to recover, e.g., by requiring some other trail of evidence such as a paper trail that can then be used instead of the – potentially buggy – software.

Finally, it should be mentioned, that a *verifiable* voting system does not automatically *verify* that it has not been tampered with. Generally, this requires also that (a) enough voters and observers must make use of the verification mechanisms and carefully check their verification result, (b) random audits (e.g., triggered by a voter complaint) must be sufficiently extensive and unpredictable such that critical changes or errors are likely to get detected, and (c) any failed check must be reported to the election authorities who then must take appropriate action, e.g., for a strongly software-independent voting system to recover the system. However, these requirements reach into the human layer in the sense that, e.g., the verifiability mechanisms must be sufficiently usable and available.

Part II

Secure Voter-Ballot Box Communication Channels

Democracy is best seen as the opportunity of participatory reasoning and public decision making - as "government by discussion." Voting and balloting are, in this perspective, just part of a much larger story.

Amartya Sen, *The Diverse Ancestry of Democracy*, 2005

4

Generation of Secure Card-Based Communication Schemes

> Lots of people working in cryptography have no deep concern with real application issues. They are trying to discover things clever enough to write papers about.
>
> Whitfield Diffie, *Foreword to Crypto 101*, 2017

Besides the many challenges that come with the development of voting systems for multiple different stakeholders and different voting channels, reliability and security are already non-trivial when voting at a smaller scale. In the following, we consider the smallest scale where we can still speak of voting, namely for two voters and a simple yes/no-decision where unanimous approval is required. We also assume no incentive to scrutinize the result on either side, but possibly to hide the individual votes to the other voter.

For a simple example, take a decision where the individual vote might be embarrassing in the event that the other voter does not vote similarly, e.g., the so-called "dating problem". The dating problem targets the scenario where two people are on their first date together and would like to find out whether to have a second date thereafter. The problem hence is the following: In case only one of them likes to meet again, it might be an uncomfortable embarrassment for any of them to reveal their choice to the other person or to learn about the other person's choice, since – in this case – the preference is not mutual. Ideally, they are interested in a discreet procedure that outputs "yes" if both share the mutual interest to have a second date, and "no" otherwise, with no more information being revealed. In

cryptography, this is called a secure multi-party computation (MPC), i.e., the act of jointly computing a function while not revealing more information about each individual input than absolutely necessary (Section 3.1). For the dating problem from above, we further assume that the two people do not want to share their individual choices with anybody else, also not a *possibly corrupted* computer. Fortunately, MPC can be already done with simple physical objects, e.g., a (regular) deck of playing cards where the cards have indistinguishable backs (which is commonly available). Card-based MPC is reasonably simple and practicable, and it can be explained to non-experts – as no special knowledge is required – and there is a *bridge to reality* (the cards).

Within this chapter, we consider card-based cryptographic protocols that perform a multi-party computation for the AND function of two bits using only the two standard card operations *turn* and *shuffle*. When combined with protocols for performing NOT and COPY functions, such simple card-based AND protocols can be used to compute arbitrary Boolean functions, e.g., also for any finite amount of voters who want to elect one of two alternatives (Mizuki, Asiedu, and Sone, 2013). The contribution consists in a formal method that automatically generates card-based AND protocols, that are secure and correct, for as few as possible card operations and a minimal amount of cards. Our method is flexibly adaptable to a variety of card-based protocols and desired restrictions, and may thus support or complement the cumbersome and error-prone task of finding such protocols or proving their non-existence by hand. We apply our method both to a setting where all cards carry distinct symbols, i.e., a standard deck, and to the more general setting where we only distinguish two symbols, e.g., the two colors ♣ and ♡, i.e., essentially a two-color deck. Finally, we also use our method to reduce computational complexity by computing the maximal size of necessary permutation sets for the shuffle operations as, e.g., for the two-color deck setting, the amount of possible distinguishable card sequences for a given protocol state is potentially significantly smaller than the amount of possible permutations on the card deck.

Our method exploits the observation that all findings in the literature employ only protocols with runs of comparatively small length using only a few cards. Our hypothesis is that we may always find some number n which is greater than or equal to the length of any run-minimal card protocol. For this scenario, we base our method on the automatic off-the-shelf technique of *software bounded model checking (SBMC)* (Section 2.2). Thereby, given such a bound n, we can encode the task into a decidable set of logical equations, which can then be solved by a SAT or an SMT solver. Based on SBMC, our method hence, given the desired numbers for amount of cards and protocol length, either constructs such a protocol if and only if one exists, or otherwise proves the underlying SAT formula to be unsatisfiable, i.e., shows that no such protocol exists.

The content of this chapter has been previously published by Koch, Schrempp, and Kirsten (2021) and Koch, Schrempp, and Kirsten (2019).

4.1 Card-Based Cryptographic Schemes

Card-based cryptography tries to find protocols, e.g., for the above-mentioned AND functionality, which are card-minimal, simple and practicable. For simplicity, many protocols in card-based cryptography work with specially constructed decks, e.g., of only the two symbols ♣ and ♡. This is easy for explanation, and there are nice and easily describable protocols, such as the five-card trick by den Boer (1989) and the six-card AND protocol by Mizuki and Sone (2009). The feasibility of card-based cryptographic MPC is due to den Boer (1989), Crépeau and Kilian (1993), and Niemi and Renvall (1998), with a formal model given by Mizuki and Shizuya (2014). Other works looking at standard decks are by Niemi and Renvall (1999) and Mizuki (2016), and other works that provide lower bound are given by Koch, Walzer, and Härtel (2015), Kastner et al. (2017), and Koch (2018) for the two-color deck setting. A card-minimal protocol for this setting that uses only practicable (i.e., uniform closed) shuffles is given by Abe et al. (2018).

4.1.1 A Simple Protocol with Five Cards

In order to get a feeling for how such card-based protocols work, let us introduce the prominent five-card protocol by Niemi and Renvall (1999). The protocol uses five cards with distinguishable symbols, which we denote – for simplicity – as $\boxed{1}$ $\boxed{2}$ $\boxed{3}$ $\boxed{4}$ and $\boxed{5}$. It is essential that the cards' backs are indistinguishable, such that when they are put face-down on the table, the only thing observable is  . With these cards, each of the two players can encode a commitment to a bit (yes or no) by the order of two cards $\boxed{i}\,\boxed{j}$, $i, j \in \{1, \ldots, 5\}$ (with $i \neq j$) via the encoding

$$\boxed{i}\,\boxed{j} \mathrel{\hat{=}} \begin{cases} 0, & \text{if } i < j, \\ 1, & \text{if } i > j. \end{cases}$$

Player one inputs their bit by putting the cards $\boxed{1}$ $\boxed{2}$ face-down and in the respective order on the table (they put $\boxed{1}$ $\boxed{2}$ for input 0, and $\boxed{2}$ $\boxed{1}$ for input 1), while player two does the same using their cards $\boxed{3}$ $\boxed{4}$. We furthermore need an additional helper card, for this matter a $\boxed{5}$, which is put to the left of the two players' cards.

The protocol starts by swapping the first player's second card with the second player's first card in the card sequence (pile) on the table. In the resulting card configuration, the order of the cards $\boxed{1}$ and $\boxed{4}$ in this sequence encodes the output of the protocol we are interested in, i.e., it reads $\boxed{4}$ $\boxed{1}$ if the output is 1, and $\boxed{1}$ $\boxed{4}$ otherwise. Hence, by securely removing the cards $\boxed{2}$ and $\boxed{3}$ (which is explained below), one directly obtains the desired output. We see this by inspecting all possible cases as shown in the table below:

Bits	Input sequence	After swap	Removing $\boxed{2}+\boxed{3}$
$(0,0)$	$\boxed{5}\,\boxed{1}\,\boxed{2}\,\boxed{3}\,\boxed{4}$	$\boxed{5}\,\boxed{1}\,\boxed{3}\,\boxed{2}\,\boxed{4}$	$\boxed{5}\,\boxed{1}\,\boxed{x}\,\boxed{x}\,\boxed{4}$
$(0,1)$	$\boxed{5}\,\boxed{1}\,\boxed{2}\,\boxed{4}\,\boxed{3}$	$\boxed{5}\,\boxed{1}\,\boxed{4}\,\boxed{2}\,\boxed{3}$	$\boxed{5}\,\boxed{1}\,\boxed{4}\,\boxed{x}\,\boxed{x}$
$(1,0)$	$\boxed{5}\,\boxed{2}\,\boxed{1}\,\boxed{3}\,\boxed{4}$	$\boxed{5}\,\boxed{2}\,\boxed{3}\,\boxed{1}\,\boxed{4}$	$\boxed{5}\,\boxed{x}\,\boxed{x}\,\boxed{1}\,\boxed{4}$
$(1,1)$	$\boxed{5}\,\boxed{2}\,\boxed{1}\,\boxed{4}\,\boxed{3}$	$\boxed{5}\,\boxed{2}\,\boxed{4}\,\boxed{1}\,\boxed{3}$	$\boxed{5}\,\boxed{x}\,\boxed{4}\,\boxed{1}\,\boxed{x}$

We can remove the cards $\boxed{2}$ and $\boxed{3}$, while keeping the relative order of all cards in the sequence intact, by cutting the cards, i.e., rotating the sequence by a random offset which is unknown to the players. We can then securely turn the first card and remove it in case it is a $\boxed{2}$ or a $\boxed{3}$. Due to the cut, the turned card is random and hence does not reveal anything about the inputs. When both cards are removed, we reach a configuration where $\boxed{5}$ is the first card by the same procedure, where the two remaining cards encode the AND result. Here, the $\boxed{5}$ plays the crucial role of a separator that keeps the relative order of the remaining cards – starting from the separator – intact, when doing a random!cut. For a pseudocode description, see Algorithm 4.1.

Algorithm 4.1 Five-card AND protocol by Niemi and Renvall (1999). The first bit is in basis $\{1,2\}$, the second in basis $\{3,4\}$, and the output basis is $\{1,4\}$.

```
1: (perm, (3 4))
2: repeat
3:     (shuffle, ⟨(1 2 3 4 5)⟩)
4:     v := (turn, {1})
5: until v = 2 or v = 3
6: repeat
7:     (shuffle, ⟨(2 3 4 5)⟩)
8:     v := (turn, {2})
9: until v = 2 or v = 3
10: repeat
11:     (shuffle, ⟨(3 4 5)⟩)
12:     v := (turn, {3})
13: until v = 5
14: (result, 4, 5)
```

As Niemi and Renvall state, the protocol's running time in the amount of shuffle steps is calculated as follows: The protocol starts with a shuffle that is then repeated with probability $3/5$. The second loop contains a shuffle that is repeated with a probability of $3/4$. The shuffle in the final loop is hence repeated with probability $2/3$. In total, the expected running time is $3 + \sum_{n=1}^{\infty}\left(\frac{3}{5}\right)^n + \sum_{n=1}^{\infty}\left(\frac{3}{4}\right)^n + \sum_{n=1}^{\infty}\left(\frac{2}{3}\right)^n = 3 + 1.5 + 3 + 2 = 9.5$. However, the last loop is actually neither required for correctness nor for security, and we

can directly determine the result when we look at the position of the $\boxed{5}$. Therefore, when we omit the last loop, we have an expected amount of $3 + 1.5 + 3 = 7.5$ shuffle steps.

4.1.2 General Card-Based Protocols

In the following, we introduce the general model for card-based protocols with some basic required notions.

Basic Notions. Formally, a *deck* $\mathcal{D}$ of cards is a multiset over a *(deck) alphabet* or a symbol set Σ. We denote multisets by $[\![\cdot]\!]$, e.g., $[\![\heartsuit, \heartsuit, \clubsuit, \clubsuit]\!]$ is a deck over $\{\heartsuit, \clubsuit\}$, and $[\![1, \ldots, n]\!]$, $n \in \mathbb{N}$, where each symbol occurs exactly once, is a deck over $[\![1, \ldots, n]\!]$, $n \in \mathbb{N}$. We call the former ones *two-color decks* and, following Mizuki (2016), the latter ones *standard decks*, because decks of common card games are a good representation of such formal decks.

A card that is lying on a table (as usual in card-based protocols) always has exactly one of two orientations, either *face-up* (showing the symbol of the card), or *face-down*. A special *back symbol* '?' that is not part of Σ represents what is visible for a card that is turned face-down. We hence describe a *card lying on the table* by a fraction symbol $\frac{a}{b}$, where exactly one of a and b has the value '?', and the other has a symbol from Σ. Here, a represents the part that is visible from the card when it lies on the table, hence $\frac{a}{b}$ is a face-down card if we have $a = ?$, and a face-up card if $a \in \Sigma$ holds. As card-based protocols usually involve some turning-over of the cards, the $\frac{a}{b}$-status will likely change during the course of a protocol, causing the numerator and denominator to be swapped.

Card-based protocols proceed on sequences of cards $(\alpha_1, \ldots, \alpha_{|\mathcal{D}|})$ where all cards from the deck $\mathcal{D}$ are lying on the table as described in the given order. Taking the "visible" numerators of all cards in the given order then yields the *visible sequence*. For example, the sequence $(\frac{?}{\heartsuit}, \frac{?}{\heartsuit}, \frac{\clubsuit}{?}, \frac{?}{\clubsuit})$ yields the visible sequence of $(?, ?, \clubsuit, ?)$. The *sequence trace* of a finite protocol run, and analogously its *visible sequence trace*, is then the sequence of all card sequences and visible sequences, respectively, as they occur during the run. We denote the set of sequences on deck $\mathcal{D}$ by $\mathsf{Seq}^{\mathcal{D}}$. Following Kastner et al. (2017), no computational power is gained by leaving cards face-up after the respective turn operation, and we safely assume that any face-down cards that are turned over during a step in the protocol are directly turned back after learning its symbol. We denote the (face-down) sequence of only the card symbols ($(\heartsuit, \heartsuit, \clubsuit, \clubsuit)$ in the example above) by the shorthand *symbol sequence*. Moreover, we assume a linear order on the card symbols in Σ, which is needed when encoding a bit. Without any loss of generality, we take here the usual order on $\mathbb{N}$ for standard decks, and $\clubsuit < \heartsuit$ for simple two-color decks.

As in the example before, two face-down cards with distinct symbols $s_1, s_2 \in \Sigma$ then *encode a bit* via the same rule:

$$s_1\, s_2 \mathrel{\hat{=}} \begin{cases} 0, & \text{if } s_1 < s_2, \\ 1, & \text{if } s_1 > s_2. \end{cases}$$

Card-based protocols proceed essentially by two actions on the sequence or pile of cards: (1) We introduce uncertainty (about which card is which) by shuffling them in an arbitrary or a certain controlled way, e.g., by cutting the cards in a quick succession, such that the players do not know which card ended up at which position in the card sequence (or pile). Formally, a (uniform) shuffle is specified by a permutation!set, from which one element is drawn uniformly at random and applied to the cards, without the players learning which one it was. (2) Secondly, we may turn over cards and publicly learn their symbol, such that we may act on the basis of this information. Moreover, we may deterministically permute the cards.

A protocol computes a Boolean function $f\colon \{0,1\}^2 \to \{0,1\}$ if the possible start sequences, corresponding to the players' inputs $b \in \{0,1\}^2$, do encode these inputs as described above, and the cards that are declared to contain the output value upon termination of the protocol do encode the output value $o = f(b)$ for each respective input $b \in \{0,1\}^2$ as described above.

Permutations and Groups. We now introduce more notation for the permutations that happen within such protocols, in order to formalize their computational model such that the security notions we are interested in can be readily expressed in a precise and compact manner within Section 4.2. Let S_n denote the *symmetric group* on $\{1, \ldots, n\}$. For the elements $x_1, \ldots, x_k \in \{1, \ldots, n\}$, the *cycle* $(x_1\ x_2\ \ldots\ x_k)$ is the *cyclic* permutation π with $\pi(x_i) = x_{i+1}$ for $1 \leq i < k$, $\pi(x_k) = x_1$ and $\pi(x) = x$ for all x not occurring in the cycle. Every permutation can be written as a composition of pairwise disjoint cycles, e.g., (1 3 2)(4 5) maps $1 \mapsto 3, 3 \mapsto 2, 2 \mapsto 1, 4 \mapsto 5$, and $5 \mapsto 4$. The identity permutation is denoted as id. Given the permutations $\pi_1, \ldots, \pi_k \in S_n$, the groups that are generated by $\pi_1, \ldots, \pi_k$ are denoted as $\langle \pi_1, \ldots, \pi_k \rangle$. A shuffle is a *random cut* if its permutation set is the group $\langle \pi \rangle = \{\pi^0, \ldots, \pi^{l-1}\}$ that is generated by a single *cyclic* permutation element π, i.e., $(x_1\ x_2\ \ldots\ x_l)$. Furthermore, a shuffle is a *random bisection cut* if its permutation set is generated by the composition of pairwise disjoint cycles of length 2. Finally, an *S_k-shuffle* is a shuffle where the symmetric group S_k is the permutation set.

4.2 Computational Model and Security Notions

While the sequence traces and notions above already fully capture the functionality and correctness of card-based protocols, the analysis and definition of their security both require a more involved model that additionally allows to distinguish different situations that are *simultaneously* reachable from a given (card) situation. For this matter, the following Section 4.2.1 describes the KWH trees as defined by Koch, Walzer, and Härtel (2015), and shown to be equivalent to the computational model by Mizuki and Shizuya (2014) and Mizuki and Shizuya (2017) in the work by Kastner et al. (2017). KWH trees compactly illustrate the tree of all reachable card situations with their respective probabilities. We then formalize their security notions in Section 4.2.2.

4.2.1 Computational Model and Protocol State Tree Representation

Let us first describe what a state during a run of a card-based protocol is. We start by an exemplary start state in Figure 4.1 of a protocol in the very beginning, i.e., after the players have put their cards that encode their inputs on the table.

12 34	X_{00}
12 43	X_{01}
21 34	X_{10}
21 43	X_{11}

Figure 4.1: The start state.

For the sake of compactness, we only write symbol sequences instead of full card sequences. Each line in the state depicted by the box above describes a card sequence that is possible at this point in time in the protocol, together with a certain type of polynomial in the variables $X_{00}, X_{01}, X_{10}, X_{11}$. For example, the first line of the state can be read as "the sequence $(\frac{?}{1}, \frac{?}{2}, \frac{?}{3}, \frac{?}{4})$ lies on the table with the symbolic probability X_{00}". Here, X_{00} is a symbolic variable, and not a concrete value, that holds the probability value that $(0, 0)$ is the input of the protocol. Note that for the depicted state, we do not have any information about the four probability values, as the input distribution can be arbitrary. As described earlier, we see here that 12 and 34 encode 0, e.g., as shown in the first line for the input $(0, 0)$ (hence the name X_{00} for the probability variable). Moreover, the order of the rows is of no significance.

We capture the formal notion of a state by the following definition:

Definition 4.1 (State in a KWH tree) *Let $\mathcal{D}$ be a deck of a protocol $\mathcal{P}$ that computes a Boolean function $f\colon \{0,1\}^2 \to \{0,1\}$. A* state *$\mu$ in $\mathcal{P}$ is a map $\mu\colon \mathsf{Seq}^{\mathcal{D}} \to \mathbb{X}_2$, where $\mathbb{X}_2$ of the form $\sum_{b\in\{0,1\}^2} \beta_b X_b$ denotes the polynomials over the variables X_b for $b \in \{0,1\}^2$,*

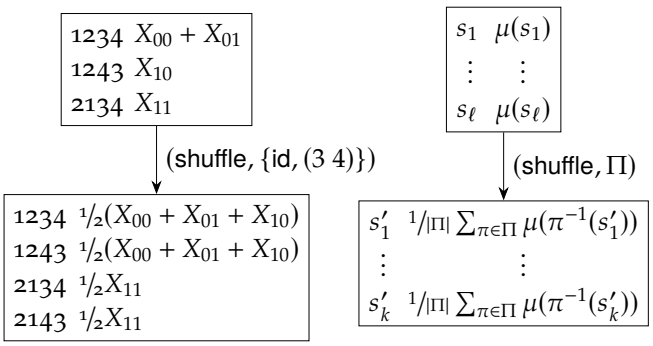


Figure 4.2: A shuffle operation, given by example (left), and the general action (right).

$\beta_b \in [0,1] \subset \mathbb{R}$. *Therein, $\mu(s)$ ($s \in \mathsf{Seq}^{\mathcal{D}}$) is interpreted as the probability, in terms of the symbolic probabilities on the inputs, that s is the actual sequence on the table.*

Such illustrative boxes are depictions of such a (state) map, where we only include exactly those sequences $s \in \mathsf{Seq}^{\mathcal{D}}$ that have a non-zero probability. We annotate each such sequence s to their right with the polynomial $\mu(s)$.

Every standard-deck protocol starts by a state as in Figure 4.1, but we possibly add further cards (5, 6, ...) if a sequence extends further to the right of the players' bits. The state tree of a protocol is then a directed tree where the nodes are states as above, where annotations at the outgoing edges of each state specify the action, i.e., card operation, that is performed next. With the state μ that has outgoing annotations, the possible actions are defined as in the following enumeration:

1. (shuffle, Π) leads to μ' as in Figure 4.2 for the permutation set $\Pi \subseteq S_{|\mathcal{D}|}$.
2. (turn, T) branches the tree into n states μ_v for each observation v possible by revealing the cards at positions from the set $T \subseteq \{1, \ldots, |\mathcal{D}|\}$, as in Figure 4.3. Therein, μ_v contains the sequences from μ which are compatible with the observation v. For each sequence s compatible with v, we have $\mu_v(s) := \mu(s)/\Pr[v]$, where $\Pr[v] \in (0,1]$ is the probability of observing v. As motivated above, we omit the implicit turn that puts the cards back face-down.
3. (perm, π) permutes the sequences of μ according to π.
4. (result, p_1, p_2) stops the computation and returns the cards at p_1, p_2 as output.

We say that a protocol computes a Boolean function $f\colon \{0,1\}^2 \to \{0,1\}$ if and only if the following two conditions *(Correctness)* hold:

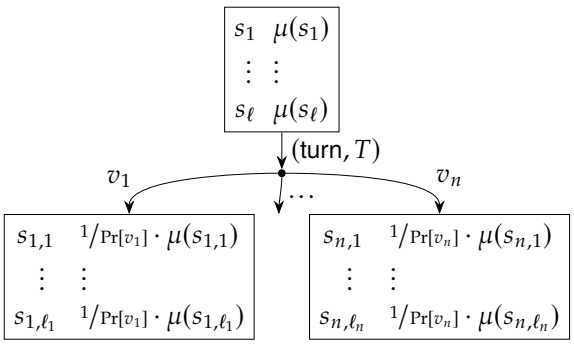


Figure 4.3: A turn operation on the cards at positions in T with the possible observation $v_1, \ldots, v_n$. For each $i \in \{1, \ldots, n\}$, $s_{i,1}, \ldots, s_{i,\ell_i}$ are the sequences from $s_1, \ldots, s_\ell$ which are compatible with v_i. In secure protocols, the probability of observing v_i, i.e., $\Pr[v_i]$, is constant.

1. the start state (the tree root) encodes each $b \in \{0,1\}^2$ in the first four cards (the remaining cards are at fixed positions), and
2. in the leaf nodes, the cards at the positions given by the result operation encode a value $o \in \{0,1\}$ if $f(i) = o$ holds for all X_i occurring in $\mu(s)$ for sequence s.

We say that a protocol has a *finite runtime* if its tree is finite. It is a *Las Vegas* protocol, if it is not finite-runtime, but the expected length of any path in its tree, i.e., the expected value of the length of an arbitrary descending path in the tree starting from the root (as a random variable, where the randomness is in the choice of the path), is finite. While we consider looping protocols, we do not consider the case which requires a complete restart. We simplify self-similar infinite trees by drawing edges to earlier states.

4.2.2 Security of Card-Based Protocols

We slightly adjust the security notion from the literature to standard decks. For more details, we refer to Koch (2019). Since different encodings for the same bit are possible, we want the encoding basis of the output bit to not give away anything about the inputs. We say that a protocol is *secure* if both at any turn operation the probability of each observation v is a constant $\rho \in [0,1]$ and at any result operation the probability of each output basis is constant in the same sense (using $\sum_{i \in \{0,1\}^2} X_i = 1$).

Moreover, we also consider a weaker form of security that is a necessary criterion for the above security notion, but still of use for the devised formal method later on. Similar to the work by Kastner et al. (2017), we define the following: A protocol is *possibilistically output-secure*, if at any state of the protocol, every output is still possible. This weakens the previous security notion, as the probability for a given input sequence could originally

have a greater value, and we could, e.g., exclude a specific input sequence in case the corresponding output is still possible through another input sequence. Together with the similar *possibilistic input-security*, we obtain the following definitions:

Definition 4.2 (Possibilistic security) *A protocol $\mathcal{P} = (\mathcal{D}, U, Q, A)$ that computes a function $f\colon \{0,1\}^2 \to \{0,1\}$ is* possibilistically input-secure (possibilistically output-secure) *if it is correct, i.e., the probability of the output being $O = f(I)$ is* 1*, and we have* $\Pr[v \mid I = i] > 0$ ($\Pr[v \mid f(I) = o] > 0$) *for uniformly*[1]*random input I, any visible sequence trace v with* $\Pr[v] > 0$*, and any input $i \in \{0,1\}^2$ (any output $o \in \{0,1\}$).*

Given such a (weaker) possibilistic security notion above, we may reduce the complexity of our state tree representation and hence of our computational model. Let us begin by defining the equivalence relation *similarity* on the states by identifying those that are only a permuted version of each other:

Definition 4.3 (Similarity) *We call two states μ and μ'* similar *if and only if there is a permutation π such that applying* (perm, π) *to μ yields μ'. The equivalence class of μ up to similarity, i.e., the set of all states that are permuted versions of μ, is called $\langle \mu \rangle_{\sim}$.*

In other words, a state μ is similar to a state μ' if μ is equal to μ' up to permutation of the columns on the sequence part of the state.

We may now define *reduced states* as in the work by Kastner et al. (2017) by omitting the state's symbolic probabilities and only including the result that is specified by their inputs, thereby greatly reducing information and state space. Any tree that contains reduced states captures a weaker form of security, namely the corresponding form of possibilistic security, where each output that is originally reachable is still reachable after reducing the state(s). Consequently, whereas a protocol that is possible for the reduced tree might not be possible for the (original) non-reduced tree, showing impossibility of a protocol for the reduced tree implies its general impossibility for the non-reduced tree.

In order to obtain a reduced state tree, we project all the symbolic probabilities of the states' sequences in a state tree to a *type* $o \in \{0,1\}$, which represents the possible future output that is associated with the sequence in a correct protocol. Let in the following $\mathcal{P}$ be a protocol that computes a function $f\colon \{0,1\}^2 \to \{0,1\}$ and μ a state in the state tree. In the first step, we set $\hat{\mu}(s) \coloneqq o \in \{0,1\}$ for any sequence s where $\mu(s)$ is a polynomial with positive coefficients for the variables $X_{b_1}, \ldots, X_{b_i}$ ($i \geq 1$), if $o = f(b_1) = f(b_2) = \cdots = f(b_i)$ holds. We refer to sequences in $\hat{\mu}$ by their type *o-sequences*. Moreover, for sequences s where $\mu(s)$ has positive coefficients for variables which represent input that would map

[1]Actually, the distribution does not matter as long as $\Pr[I = i] > 0$ holds for all $i \in \{0,1\}^2$.

to different output, e.g., $X_{00} + X_{11}$ for $f(0,0) \neq f(1,1)$,[1] we introduce the *additional type* $\perp$. Therefore, we can define a reduced state as follows.

Definition 4.4 (Reduced state) *Let $\mathcal{P}$ be a protocol that computes a Boolean function $f\colon \{0,1\}^2 \to \{0,1\}$ with the deck $\mathcal{D}$. Then a* reduced state $\hat{\mu}$ *of $\mathcal{P}$ is a map $\hat{\mu}\colon \mathsf{Seq}^{\mathcal{D}} \to \{0,1,\perp\}$ which maps a sequence $s \in \mathsf{Seq}^{\mathcal{D}}$ to its type o-*sequences.

If μ is a (non-reduced) state of $\mathcal{P}$, we can map it to its reduced state as follows: We define the reduced state $\hat{\mu}$ *of $\mathcal{P}$* based on μ *as $\hat{\mu}(s) \coloneqq t_s$, where t_s is $\mu(s)$'s type. Note that it is always possible to map a state to its reduced version.*

As an example, let us look at the following tree excerpt in Section 4.2.2 on the left of Figure 4.2, and its reduced version (here, shown on the right), where we assume it to be part of a protocol that computes the AND function:

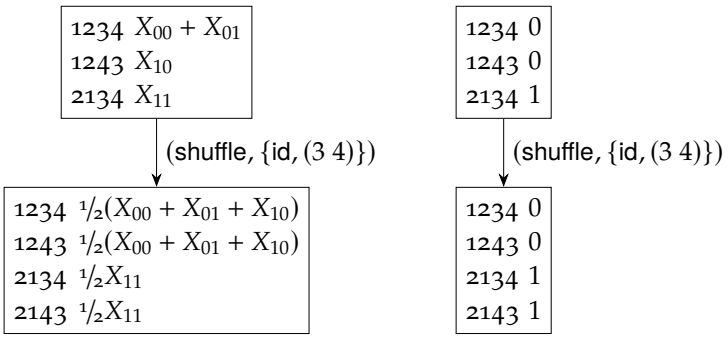


Figure 4.4: Excerpt of reduced shuffle, given byexample (left), and general action (right).

Here, the annotation of 1234 in the first state, $X_{00} + X_{01}$, is mapped to its type 0, as it only contains variables that represent inputs (namely $(0,0)$ and $(0,1)$) for the output 0. Note that by using reduced states, we bring the state space from the countably infinite to the finite, which greatly reduces proof complexity.

Finally, what remains to be defined are the situations for which the turn operation can be applied in a reduced state. A reduced state is *turnable at position $i \in \{1,\dots,|\mathcal{D}|\}$*, if for each symbol $c \in \Sigma$, there exists, among the sequences s with c at position i, an r-sequence for each $r \in \{0,1\}$ in the image of the computed function, and/or a $\perp$-sequence. Consequently, all outputs are still possible after a turn at position i, which captures the notion of output-possibilistic security. The reduced state is hence *turnable* if it is turnable at a position $i \in \{1,\dots,|\mathcal{D}|\}$.

[1] It is clear that for a state with a $\perp$ sequence, the protocol must abort later, as – if this sequence would actually lie on the table – it is no longer clear whether an input sequence for $(0,0)$, or an input sequence for $(1,1)$ was on the table at the start.

4.2.3 Two-Color Deck Protocols

As we motivated in Section 4.1 and the introduction of this chapter, the more natural (and more general) setting for card-based protocols is that of two-color decks, e.g., the two symbols ♣ and ♡. For the two-color deck setting, a card-minimal Las Vegas AND protocol using only four cards was given by Koch, Walzer, and Härtel (2015). While they use only closed shuffles, some shuffles are non-uniform and hence, the protocol is rather difficult to implement. However, it is still insightful to analyze whether the protocol features a shortest run. For this, let us note that there are two possible versions of this protocol: by contracting two subsequent closed shuffles, we can generate a protocol with fewer but non-closed shuffles. Both protocols are given in Figure 4.5 and Algorithm 4.2, where $\Pi_1, \mathcal{F}_1, \Pi_2, \mathcal{F}_2$ are permutation groups and probability distributions as follows:

$$\begin{aligned} \Pi_1 &\coloneqq \langle (1\ 2)(3\ 4) \rangle, \quad \mathcal{F}_1 \colon \mathsf{id} \mapsto 1/3, (1\ 2)(3\ 4) \mapsto 2/3, \\ \Pi_2 &\coloneqq \langle (1\ 3)(2\ 4) \rangle, \quad \mathcal{F}_2 \colon \mathsf{id} \mapsto 1/3, (1\ 3)(2\ 4) \mapsto 2/3, \end{aligned} \tag{4.1}$$

and $\alpha_1, \alpha_2, \alpha_3$ are placeholders actions as follows for the full protocol:

$$\begin{aligned} \alpha_1 &\coloneqq (\mathsf{shuffle}, \langle (1\ 3)(2\ 4) \rangle); (\mathsf{shuffle}, \langle (2\ 3) \rangle), \\ \alpha_2 &\coloneqq (\mathsf{shuffle}, \langle (1\ 3) \rangle); (\mathsf{shuffle}, \Pi_1, \mathcal{F}_1), \\ \alpha_3 &\coloneqq (\mathsf{shuffle}, \langle (3\ 4) \rangle); (\mathsf{shuffle}, \Pi_2, \mathcal{F}_2), \end{aligned} \tag{4.2}$$

and for the protocol using contracted shuffles as below:

$$\begin{aligned} \alpha_1 &\coloneqq (\mathsf{shuffle}, \{\mathsf{id}, (1\ 3)(2\ 4), (2\ 3), (1\ 2\ 4\ 3)\}), \\ \alpha_2 &\coloneqq (\mathsf{shuffle}, \{\mathsf{id}, (1\ 3), (1\ 3)(2\ 4), (1\ 4\ 3\ 2)\}, \mathcal{F}_3), \\ \mathcal{F}_3 &\colon \mathsf{id} \mapsto 1/6, (1\ 3) \mapsto 1/6, (1\ 3)(2\ 4) \mapsto 1/3, (1\ 4\ 3\ 2) \mapsto 1/3, \\ \alpha_3 &\coloneqq (\mathsf{shuffle}, \{\mathsf{id}, (3\ 4), (1\ 3)(2\ 4), (1\ 3\ 2\ 4)\}, \mathcal{F}_4), \\ \mathcal{F}_4 &\colon \mathsf{id} \mapsto 1/6, (3\ 4) \mapsto 1/6, (1\ 3)(2\ 4) \mapsto 1/3, (1\ 3\ 2\ 4) \mapsto 1/3. \end{aligned} \tag{4.3}$$

4.3 Trace-Based Formal Security Verification

Based on the KWH trees (Koch, Walzer, and Härtel, 2015) introduced in the previous section, we can formalize a standardized program representation to be instrumented by our formal method.

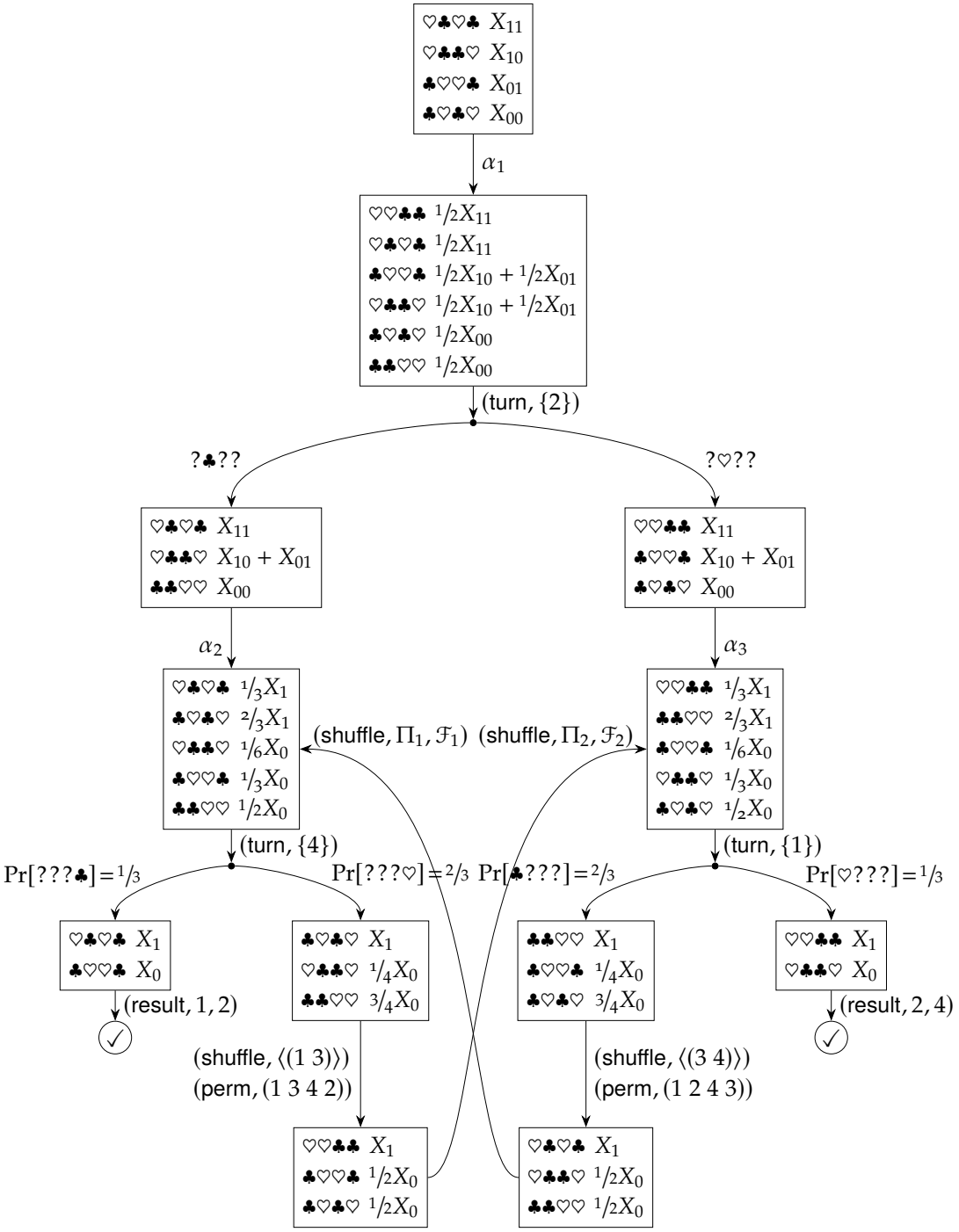


Figure 4.5: The four-card protocol by Koch, Walzer, and Härtel (2015), with placeholders as specified in the text, to define two similar variants of the same protocol. The contracted, non-closed variant has a shortest run of length 4, while the closed variant has a shortest run of length 6.

Algorithm 4.2 Two protocols to compute AND using four cards, cf. also Figure 4.5. The placeholders Π_i, $\mathcal{F}_i$ are given in Equation (4.1) and the α_i are defined in Equations (4.2) and (4.3).

```
 1: α_1
 2: (turn, {2})
 3: if v = (?, ♣, ?, ?) then
 4:     (turn, {2}) // turn back
 5:     α_2
 6:     (turn, {4})
 7:     if v = (?, ?, ?, ♣) then
 8:         (result, 1, 2)
 9:     else if v = (?, ?, ?, ♡) then
10:         (turn, {4}) // turn back
11:         (shuffle, {id, (1 3)})
12:         (perm, (1 3 4 2))
13:         (shuffle, Π_2, F_2)
14:         goto 6
15:     end if
16: else if v = (?, ♡, ?, ?) then
17:     (turn, {2}) // turn back
18:     α_3
19:     (turn, {1})
20:     if v = (♡, ?, ?, ?) then
21:         (result, 2, 4)
22:     else if v = (♣, ?, ?, ?) then
23:         (turn, {1}) // turn back
24:         (shuffle, {id, (3 4)})
25:         (perm, (1 2 4 3))
26:         (shuffle, Π_1, F_1)
27:         goto 19
28:     end if
29: end if
```

4.3.1 Standardized Program Representation

A standardized program representation allows a general programmatic encoding of both the shuffle and the turn operation, as well as of the fixed input state (indicated by the input card sequences from the table in Figure 4.1), the nondeterministic reachable states, and the logical function to be computed securely. The input state is trivially derived from the specified amounts of cards as the size and order of the players' commitments is fixed and the (without loss of generality) consecutively ordered card sequence of (distinguishable) helper cards is simply prepended to the input card sequence, annotated with their respective input probabilities. Any input state for Boolean functions thus consists of exactly four distinguishable card sequences.

Based on this input state, the program performs the loop that is illustrated in Algorithm 4.3, which successively performs turn or shuffle operations based on the input state and computes the resulting states from which it continues performing turn or shuffle operations. The loop ends when the specified bound L (representing the length of the protocol to be found) is reached, checks whether the final state is indeed a valid computation of the secure function, and (if and only if the check is successful) the found protocol – represented by a program trace that lists the traversed program states – is then presented to the user.

Algorithm 4.3 Overall program loop that models the nondeterministic choice of a succession of actions in the protocol up to L steps.

Input:

reachableStates: list that collects the reachable states, initialized with the start state.
L: maximum length of the protocol.
LAS_VEGAS: parameter that is true only for Las Vegas protocols.
MAX_TURN_OBSERVATIONS: maximum distinguishable observations for one card.

Output:

A Boolean value that is TRUE if and only if a valid protocol exists within the bounds.

```
1:  function performProtocol
2:      for i ← 0 to L − 1 do
3:          action ← nondet_choice(TURN, SHUFFLE)
4:          if action = TURN then
5:              possiblePostStates ← applyTurn(reachableStates[i])
6:              turnPosition ← nondet_choice(0, ..., MAX_TURN_OBSERVATIONS − 1)
7:              if not isObservable(possiblePostStates[turnPosition]) then
8:                  return FALSE
9:              reachableStates[i + 1] ← possiblePostStates[turnPosition]
10:             if (LAS_VEGAS and isFinalState(reachableStates[i + 1])) or
11:                 (not LAS_VEGAS and isFinalTurn(possiblePostStates)) then
12:                 return TRUE
13:         else if action = SHUFFLE then
14:             reachableStates[i + 1] ← applyShuffle(reachableStates[i])
15:             if isFinalState(reachableStates[i + 1]) then
16:                 return TRUE
17:         end if
18:     end for
19:     return FALSE
20: end function
```

This task involves multiple computational complexities, most notably both the amount of (possibly) reachable states, and the choice of the next operation, i.e., either choosing the card(s) to be turned or which shuffle to perform. We partially overcome the first computational complexity by not considering Las Vegas protocols, as this relieves us from checking every reachable sequence of states to be finite. In fact, we compute all reachable states after every protocol operation, but only check each of them to be valid, and then proceed our operations on only one of them, which is nondeterministically (denoted by `nondet_choice`) chosen among them. The second computational complexity consists in first nondeterministically choosing whether to shuffle or to turn, and then to perform the respective operation.

The turn!operation consists of a mostly obvious implementation for updating the computed state and its probabilities using mostly standard imperative program operations. Note that the turn observations are again nondeterministically chosen, hence considering any of them to be possible. Moreover, the shuffle operation must randomly draw a set of permutations on which the reachable states are computed. We implement this by nondeterministically choosing a set of permutations from a precomputed set of all permutations that are generally possible. Both the amount and the choices of the respective permutations are chosen nondeterministically.

4.3.2 Verification Methodology

The above generic program representation of KWH trees together with implementations for the start state and further operations, such as the methods that are used therein including the turn and the shuffle action, can now be instrumented by a software bounded model checker (SBMC) (Section 2.2).

For this matter, after iterating the aforementioned loop for the specified bound with the described operations and restricting that the final state indeed computes the secure function, we specify the software property C^{cons} to be checked such that `performProtocol` must return the Boolean value `false`. This property implies that the verification task always fails once there exists input and nondeterministic parameters such that the respective program run reaches the statement in the program which checks this property, if the protocol generated by Algorithm 4.3 is valid (i.e., returns the value `true`). The SBMC tool exhaustively searches for a run of the specified length through the program which leads from the starting state to a correct and secure state which satisfies the given security notion, i.e., reaches the above-mentioned statement. Hence, if there exists any protocol of the specified length which computes the secure function and for which the specified operations and valid intermediate states (representing KWH-trees) exist, such a protocol is presented by our method. If no such protocol can be found, we know there is no card-based protocol of the specified length satisfying all our restrictions on permitted

turn and shuffle operations, as well as intermediate and final states. In this case, there exists no model for the SAT formula which encodes the set of all permitted program runs given our specified requirements.

Hence, assuming our translation of KWH trees and respective protocol operations into a simple imperative program are correct, this method can then be used iteratively to generate (possibilistically-)secure card-based protocols of the given length or prove that none exists within the specified bounds. That mechanism can be useful either for finding new protocols or tighten the bounds for run-minimal protocols from the literature. Software bounded model checking draws its purpose largely from the so-called *small-scope hypothesis*, i.e., that a large amount of bugs is already exposed for small program runs. We apply this hypothesis to the setting of card-based security protocols, as all protocols in the literature only use a few turn and shuffle operations and the length of any found protocol is below ten operations.

This approach can be generalized to search for card-based protocols using a predefined amount of actions and adhering to a given formal security notion. We have written a general program[1] to search for such situations parameterized in the desired restrictions on actions and security notions. Note that, in order to cope with the still considerable state space size, we use the refined security notion of output-possibilistic security.

Moreover, we have the ability to restrict our experiments to only closed shuffles, and can also bound the shuffle set size to keep the complexity manageable for the verification, if needed (albeit possibly reducing the strength of the results, cf. Section 4.5).

For example, in our analysis of the run-minimality of Algorithm A.1, we bound the permitted size of the permutation sets by the – rather reasonable – number 8, in order to keep the running times still manageable for our experiments. Note that the technique described in Section 4.5 shows that only a bound of 12 would be really safe to assume, leaving a (small) gap in the argumentation as we superficially exclude exactly the possible 12-element alternating groups A_4 as shuffle steps from the possible protocol candidates.

4.4 Generation of Provenly Run-Minimal Schemes

In the following, we exemplify our translation of card-based cryptographic AND protocols using standard decks to the bounded model checker CBMC, which takes programs in the C language. For our experiments, we used CBMC 5.11 with the built-in solver, based on the SAT solver MiniSat 2.2.0 (Clarke, Kroening, and Lerda, 2004; Eén and Sörensson, 2003). All experiments are performed on an AMD Opteron(tm) 6172 CPU at 2.10 GHz with 48 cores and 256 GB of RAM.

[1]The source code is available under https://github.com/mi-ki/cardCryptoVerification.

```
1 struct sequence {
2   uint val[numberOfCards];
3   struct fractions probs;
4 };
```

Listing 4.1: C struct holding the state trees.

We translate KWH trees in the C language using a simple encoding into a bounded C program with only static structures and no pointers, e.g., we employ C structs (see Listing 4.1) holding an array of card sequences for the sequence s, attached with their respective values for each probability (for the probabilistic security notion) or dependency (for output-possibilistic security) X_i occurring in $\mu(s)$, which is encoded by another C struct `fractions`. The sequences are constructed using nondeterministic values restricted by respective software conditions to enforce a lexicographic ordering. Moreover, we assign the starting values in $\mu(s)$ with fixed (i.e., deterministic) values based on the constructed sequences. Subsequently, an array of (consecutively) reachable states is constructed nondeterministically using simple implementations of the turn and the shuffle operation as explained in Section 4.1. We then repeatedly (after each turn/shuffle) check whether all possible resulting (nondeterministic) states correctly and securely compute the specified function, e.g., within this thesis a secure AND function.

An exemplary shuffle operation is shown in Listing 4.2 for the case of output-possibilistic security. Therein, the keyword `__CPROVER_assume` is used by the bounded model checker to restrict all program runs passing this statement to satisfy the specified (Boolean) condition. By assigning values using the special function `nondet_uint()`, we assign a nondeterministic non-negative integer number, which is restricted to values greater than zero and at most of the value `NUM_POSS_SEQ` (which is a variable computed by the preprocessor and is the maximum amount of sequences possible with the given deck) in the following program statement. In the shown example, the nondeterminism is used to construct a set of permitted permutation sets (to be used by the shuffle operation), which makes the SBMC tool inspect the following program code for all possible assignments of this value. If necessary, this may result in a fully exhaustive search, however, the prover is often able to restrict the domain based on further program statements and dependencies seen in the rest of the program. A similar trick is used when computing the concrete permutations using the nondeterministic value of `permIndex`, in order to check all possible permutations which possibly move the values, but preserve all existing numbers in the sequence itself. This is done using the **`int`**-array `takenPermutations`, which is first initialized to zero and, when choosing a concrete permutation, assumed to be zero at position `permIndex`, however set to the number one right afterward (such that it is not permitted to be chosen again). In the subsequent inner loop, the permutations are assigned by choosing the according cards from the sequences in the start state using the nondeterministic value `permIndex`.

Finally, the shuffle is applied, resulting in the state variable `result`, which is then checked using a further method `isBottomFree` to not contain any sequences with impermissible values for X_i, which would result in incorrect computations of the AND function.

```
uint permSetSize = nondet_uint();
__CPROVER_assume (0 < permSetSize);
__CPROVER_assume (permSetSize <= NUM_POSS_SEQ);
uint permutationSet[permSetSize][numberOfCards];
uint takenPermutations[NUM_POSS_SEQ] = { 0 };

for (uint i = 0; i < permSetSize; i++) {
  uint permIndex = nondet_uint();
  __CPROVER_assume (permIndex < NUM_POSS_SEQ);
  __CPROVER_assume (!takenPermutations[permIndex]);

  takenPermutations[permIndex] = 1;
  for (uint j = 0; j < numberOfCards; j++) {
    permutationSet[i][j] =
      startState.seq[permIndex][j] - 1;
  }
}
struct state result =
  doShuffle(startState, permutationSet, permSetSize);
__CPROVER_assume (isBottomFree(result));
```

Listing 4.2: Simplified shuffle operation for CBMC.

4.4.1 Adaptations for Two-Color Decks

As the program by Koch, Schrempp, and Kirsten (2019) is already very general, the adaptions for covering the two-color deck settings require only little changes. In the following, we describe the changes for our verification method. The programs mainly differ in the assignment of the start state, in the following code snippets identified by `start`, for the protocol. In the following, the variable `NUM_SYM` specifies the amount of distinct card symbols, which is not needed in the standard deck setting, as there it is identical to the total amount of cards. In Listing 4.3, the variable `N` specifies this total amount of cards.

In the standard deck setting, each player gets distinct symbols 1 and 2, or 3 and 4, respectively (as shown in the first two lines in Listing 4.3). For the two-color deck setting, it suffices to require that the individual cards for each player are pairwise distinct, as shown in the first two lines in Listing 4.4. Moreover, we simply number the helper cards consecutively for the standard deck setting (see the loop in Listing 4.3), but allow an

```
1 __CPROVER_assume ((i != 0 && i != 1) || start[i] == 1 || start[i] == 2);
2 __CPROVER_assume ((i != 2 && i != 3) || start[i] == 3 || start[i] == 4);
3 for (uint i = 4; i < N; i++) {
4     start[i] = i + 1;
5 }
```

Listing 4.3: Simplified start sequence assignment in the standard deck for CBMC.

```
1 __CPROVER_assume (start[1] != start[0]);
2 __CPROVER_assume (start[3] != start[2]);
3 for (uint i = 4; i < N; i++) {
4     start[i] = nondet_uint();
5     __CPROVER_assume (0 < start[i]);
6     __CPROVER_assume (start[i] <= NUM_SYM);
7 }
```

Listing 4.4: Simplified start sequence assignment in the two-color deck for CBMC.

arbitrary assignment of valid card symbols in the two-color deck setting (see the loop in Listing 4.4).

Besides the introduction of the variable `NUM_SYM`, these are the main changes that are needed in order to cover the two-color deck setting. Note that we moreover adapt the script that calls the SBMC tool together with our C program to compute the new amount of possible sequences. For the standard deck setting, the amount is simply the factorial of the total amount of cards. In the two-color deck setting, this is the binomial coefficient of the two different amounts of cards with distinct symbols.

4.4.2 Verification Results

For standard decks, we apply our approach to the computation of a secure AND protocol using four cards in order to, firstly, substantiate our proof that no protocol of a length below six can be found, and, secondly, automatically find a permitted protocol using six operations.

For the two-color deck setting, we show by formal verification that the closed AND protocol variant has a shortest run of 6 steps, relative to all closed four-card AND protocols. This is because our method excludes the possibility of an input-possibilistic[1] closed four-card AND protocol with a run of length 5. Moreover, our contracted AND protocol is run-

[1]Because it found a possible output-possibilistic (but not input-possibilistic) protocol run, we had to strengthen the search criteria to protocols which are at least input-probabilistic.

minimal in the sense that no (output-possibilistic) four-card AND protocol with a run of length 3 exists.

For both kinds of decks, the running times and formula sizes (i.e., amounts of variables and clauses) generated by our method, we refer to Table 4.1 below.

Table 4.1: Running times for showing/disproving protocol existence for standard and two-color decks. While all rows having "✓" in the column "Protocol" indicate that a protocol run is output by our method with the CBMC running time as indicated in the table, these do not automatically feature probabilistic security. Hence, we add references to protocols in Chapter A with the given parameters, which should not (generally) be understood as having been discovered using our method.

#Cards	Shuffles	#Steps	Protocol	#Var.	#Clauses	Time
Standard Decks						
4	closed	5	✗[a]	67.3 M	266.4 M	114.1 h
4	closed	6	✓, also Figure A.1	68.2 M	269.7 M	45.3 h
Two-color Decks						
4	–	3	✗	5.2 M	20.3 M	46 min
4	–	4	✓, also Figure 4.5 with Equation (4.3)	6.9 M	27.0 M	50 min
4	closed	5	✗[b]	12.3 M	47.2 M	7.9 h
4	closed	6	✓, also Figure 4.5 with Equation (4.2)	9.3 M	34.4 M	45 min
5	closed	4	✓, also Figure A.2	22.3 M	87.2 M	45 min

[a] This holds only with respect to protocols with shuffle size of at most 8, excluding subgroups of size 12.
[b] For this, we had to strengthen the security to input-possibilistic security.

4.5 Verification of Shuffle Set Maximality

In the following, we explore the fact that the amount of possible sequences in a protocol state may be significantly smaller than the amount of possible permutations on the deck, especially for the case of two-color decks. In order to reduce the complexity of our method for finding run-minimal protocols, we adapt our formal method to additionally establish a formal guarantee that it suffices to search protocols with a smaller permutation set size, i.e., the shuffle set size. Hence, the amount of possible shuffles gets significantly smaller, which reduces the work for the SBMC tool and thus leads to significantly smaller running times.

We can write a simple program – by some simple adaptions from the program in Section 4.4 – that serves as an input for the SBMC tool to verify the maximality of a given shuffle set size. The shuffle operation from Listing 4.2 is adapted such that we can specify a lower bound for the nondeterministic variable `permSetSize`. We search for a single shuffle operation such that a valid output state is reached from a *minimal state*, i.e., a state that

```
1 uint seqIdx1 = nondet_uint();
2 uint seqIdx2 = nondet_uint();
3 __CPROVER_assume (seqIdx1 < seqIdx2);
4 minState.sequence[seqIdx1].probs = {1, 0}; // set probability to X_0
5 minState.sequence[seqIdx2].probs = {0, 1}; // set probability to X_1
6
7 struct state nextState = performShuffle(minState);
8 uint foundValidState = isValid(nextState);
9 assert (foundValidState);
```

Listing 4.5: Simplified maximality verification for CBMC.

has at most one 1-sequence and one 0-sequence (that should not be mixed together in the shuffle). The main procedure of this program is shown in Listing 4.5. Therein, we set the probabilities of two arbitrary distinct sequences in that state to be the inverse of each other, i.e., (1 0) and (0 1). We then check whether, after performing a shuffle operation on this state, we can still reach a valid state. Note that, since we are looking for worst-case maximality bounds, it suffices to employ the output-possibilistic setting (see Definition 4.2) which again reduces the search complexity.

For the verification of a maximal shuffle set size, we can run the SBMC tool on this program for various lower bounds for `permSetSize` until we find the smallest value such that no valid state is reachable anymore. This gives us a guarantee that larger shuffle set sizes cannot produce smaller protocol runs, and we can hence use this value for an upper bound on the shuffle set size in the approach from Section 4.2.3.

The described functionality in the C program is shown in Listing 4.5. Therein, `seqIdx1` and `seqIdx2` are the nondeterministically chosen indices for the zero- and one-sequence, which are assumed to be distinct. The minimal start state is given by the variable `minState` (which contains an array of sequences). We perform a nondeterministic shuffle operation on `minState` by calling the method `performShuffle`. Finally, we ask the SBMC tool to check whether the produced `nextState` is a valid state using the final `assert` statement.

Note that the results of this section in determining the maximal useful shuffle set size hold not only for AND but also for *any Boolean function* that has at least two possible outputs. The results are summarized in Table 4.2.

As an example, see Figure 4.6 (left) for the maximal shuffle set size (of 12) that is useful in four-card two-color protocols in general. Here, the shuffle starts from a minimal 2-sequence state that was chosen arbitrarily and nondeterministically by our SAT solver, but is likely to have maximal Hamming distance among their sequences. For protocols

[1]See https://groupprops.subwiki.org/wiki/Subgroup_structure_of_symmetric_group:S4 or https://groupprops.subwiki.org/wiki/Subgroup_structure_of_symmetric_group:S5, respectively, for reference.

Table 4.2: Running times for proving shuffle set size maximality. For some settings with closedness requirement, we specify ranges, which should indicate that the larger range is already impossible due to the size restrictions of subgroups.[1]

#Cards	Shuffles	Shuffle Size	Valid Shuffle	#Var.	#Clauses	Time
STANDARD DECKS						
4	–	12	✓, cf. Figure 4.7	5.2 M	12.9 M	51.9 min
4	–	13	✗	13.8 M	55.9 M	2.4 h
4	closed	12	✓, cf. Figure 4.7	13.5 M	54.0 M	16.9 min
4	closed	13–24	✗[a]	–	–	–
TWO-COLOR DECKS						
4	–	12	✓, cf. Figure 4.6	1.5 M	6.1 M	54 sec
4	–	13	✗	1.6 M	6.5 M	70 sec
4	closed	8	✓, cf. Figure 4.6	1.4 M	5.0 M	69 sec
4	closed	(9–) 12	✗	2.2 M	8.2 M	3.2 min
5	–	48	✓	13.9 M	57.0 M	3.4 h
5	–	49	✗	14.2 M	58.1 M	11.4 h
5	closed	12	✓	4.9 M	18.9 M	26.1 min
5	closed	20	?[b]	9.1 M	35.4 M	–
5	closed	24	?[b]	11.8 M	46.4 M	–
5	closed	25–120	✗[c]	–	–	–

[a] As the largest proper subgroup is of size 12, there is nothing to show (S_4 creates $\bot$-sequences).
[b] This run did not finish in time, or ran into the self-set timeout bound of 5 days.
[c] >48 permutations is impossible even non-closed, and 60 is the only *proper* subgroup size >24.

using only *closed* shuffles, our method showed that this bound is 8 permutations, as there is no larger closed permutation set that can result in a valid state, cf. Figure 4.6 (right). These bounds are fully tight.

In the *five-card* two-color deck setting, closed protocols can make use of shuffle groups of at most 24 permutations. It is an open question whether this is a tight bound, but we know that there is a 12-element shuffle that is valid. However, it still allows us to restrict the maximal shuffle group size to 24 when searching protocols. For this five-card case and arbitrary non-closed shuffle sets, the maximal shuffle set size that does not introduce $\bot$-sequences on a minimal state is 48. This is a tight bound.

Additionally, we have adapted this method to the standard deck setting as well and determined that the largest permutation set permissible in a protocol on four cards is 12. This also holds for the closed case, i.e., there is a group with 12 elements, namely the alternating group A_4, that, if performed on a minimal state, can result in a state that does not contain any $\bot$-sequences.

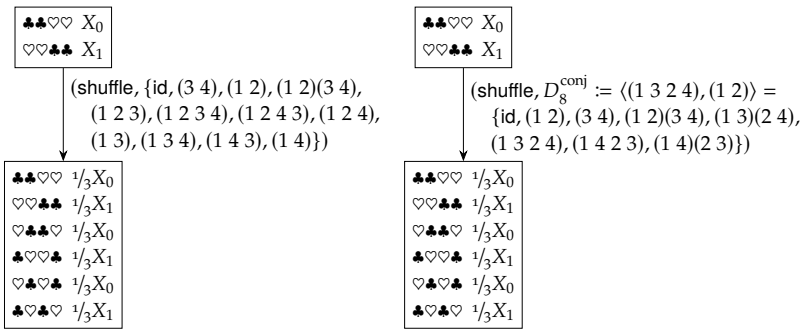


Figure 4.6: Situation discovered by our formal method to find a minimal state and a maximal permutation set (of size 12 (left) and 8 (right), respectively), such that applying this shuffle to the minimal state does not generate an invalid state (with $\perp$-sequences). Our method showed that larger shuffle sets (left) or groups (right) cannot result in valid states, allowing us to reduce the shuffle set size in verification steps without losing generality. Here, D_8^{conj} denotes a dihedral group of order 8.

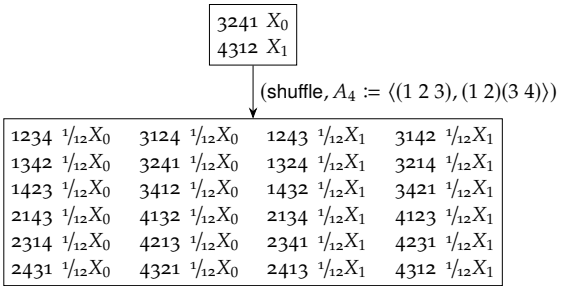


Figure 4.7: Situation discovered by our formal method to find a minimal state and a maximal permutation set (of size 12, namely the alternating group A_4), such that applying this shuffle to the minimal state does not generate an invalid state (with $\perp$-sequences), in the standard deck setting.

4.6 Summary

In this chapter, we proposed a new method to search card-based protocols for any secure computation, by giving a general formal translation applicable to be used by the formal technique of software bounded model checking (SBMC). This method allows us to find new protocols automatically, and prove lower bounds on required shuffle and turn operations for any protocol, and provide an example for the computation of a minimal AND protocol. We also found a new protocol that only uses the theoretical minimum of four distinguishable cards for an AND computation. Moreover, we supported this finding by our automatic method in showing the impossibility of any protocol using less shuffle and turn operations using only practicable shuffles (random cuts). The protocol is hence optimal with respect to the running-time restriction *restart-free Las-Vegas*. For

the four-card standard deck setting, we showed that there is no finite-runtime protocol, regardless of the shuffle operations used. This result completes the picture of tight lower bounds for the four-card setting.

Finally, we extended our verification method to the case of decks using only two colors, which is more common in the field of card-based cryptography. In this setting, we were able to show two variants of a card-minimal Las Vegas AND protocol to be also run-minimal, i.e., the protocol has a run of minimal length. Moreover, for the case of 4 cards, we derived tight upper bounds on the size of the maximal usable permutation set, of 12 and 8 for general and closed protocols, respectively. As this is not restricted to AND protocols, but applies more generally, we believe this to be of independent interest for researchers in the field of card-based cryptography.

Outlook. Let us point out some open challenges in the card-based security area that could be approached based on the findings in this chapter: (1) For finite-runtime protocols, there exist no proven tight lower bounds on the required amount of cards (five to eight cards). We recommend more research applying computer-aided formal methods at this point, as the state space for five or more cards is very large. (2) The two most common settings in card-based cryptography are the standard deck setting with only distinguishable cards and the two-color decks using ♣ and ♡. However, it may be possible that by mixing these settings (e.g., only distinguishable cards with one pair of identical cards), we might find more efficient protocols (especially in the finite-runtime setting). For such a mixed setting, Shinagawa and Mizuki (2019) provide nice results to use in further research.

Part III

Reliable Election Methods

[…] they began running when they liked, and left off when they liked, so that it was not easy to know when the race was over. However, when they had been running half an hour or so, and were quite dry again, the Dodo suddenly called out "The race is over!" and they all crowded round it, panting, and asking, "But who has won?"

Lewis Carroll, *Alice's Adventures in Wonderland*, 1865

5

Synthesis of Reliable Tallying Procedures

> Simplicity is prerequisite for reliability.
>
> Edsger Wybe Dijkstra, *How Do We Tell Truths that Might Hurt?*, 1975

This chapter presents a formal systematic approach for the flexible and verified synthesis of tallying procedures from compact composable modules with guaranteed formal properties. The formal properties for this matter are social choice properties as introduced in Section 3.2 and can be understood as the characteristics required by the election method, e.g., legal regulations. By tallying procedures, we understand the notion of voting rules from social choice theory (Definition 3.1 in Section 3.2).

Indeed, there is no general rule which caters for every requirement, and any voting rule shows paradoxical behavior for some voting situation (Arrow, 1950). Therefore, an approach to analyze voting rules for their behavior by clear-cut formal properties, the so-called *axiomatic method*, has emerged. The axiomatic method advocates the use of rules that provide rigorous guarantees (which we call *properties*), and compares them based on guarantees that they do, or do not, satisfy. These properties capture very different requirements of fairness or reliability, e.g., principles that each vote is counted equally, that the electees proportionally reflect the voters' preferences, or that they are preferred by a majority of the voters. Devising voting rules towards such properties is generally cumbersome, as their tradeoff is inherently difficult and error-prone. Attempting to prove properties for specific voting rules often exhibits design errors, but is cumbersome as well (Beckert, Goré, and Schürmann, 2013). As of yet, there exists no general formal

approach to systematically devise voting rules towards formal properties without being either error-prone or extremely cumbersome.

When taking an abstract view, many voting rules share similar structures, e.g., aggregating the individual votes by calculating the sum or some other aggregator function. Based on this observation, our approach enables flexible, intuitive and verified synthesis of interesting voting rules from a few compositional structures. These structures exhibit precise and general interfaces, such that their scope may easily be extended with further modules. We devise a general component type inspired by Grilli di Cortona et al. (1999) as well as special types, e.g., for aggregation functions, and compositional structures, e.g., for sequential, parallel and loop composition, together forming our construction framework. The resulting properties, e.g., common social choice properties from the literature, are guaranteed from composing modules with given individual properties by rigorous composition rules.

We demonstrate the logic-based application with proofs for a selected set of composition structures and rules, and composable modules within the theorem prover Isabelle/HOL (Nipkow, Paulson, and Wenzel, 2002). Thereby, the approach is amenable both for external scrutiny as compositions are rigorously and compactly defined, and for an integration in larger automatic voting rule design or verification frameworks. As a case study, we define composition rules for the common social choice property *monotonicity*, and demonstrate a formal correct-by-construction synthesis of the rule *sequential majority comparison* (SMC). The synthesis produces a proof that SMC fulfills the monotonicity property using a set of basic modules. We further extend our framework with an abstract module for constructing various practical Condorcet rules from literature and proving that they are indeed Condorcet consistent. We finally built an automatic synthesis tool that instruments our framework and, given the desired social choice properties, automatically synthesizes a suitable voting rule together with an automatic generation of the Isabelle proof and translation of the voting rule into executable software.

The content of this chapter has been previously published by Diekhoff, Kirsten, and Krämer (2020) and Diekhoff, Kirsten, and Krämer (2019), extended with Condorcet rules from the bachelor's thesis by Bohr (2020), and an automated synthesis from the master's thesis by Richter (2021).

5.1 Composition of Voting Rules

The construction framework consists of two structural and two semantic concepts, namely (i) *component types* that specify structural interfaces wherein components can be implemented, and (ii) *compositional structures* that specify structural contracts which combine components to create new components that are again composable. Moreover, semantic

aspects for constructing concrete voting rules are addressed by (iii) *composition rules* that define semantic rules which compositions can contractually depend on, i.e., if components fulfill a rule's requirements, the composition guarantees the rule's semantics, as well as (iv) *composable modules* that define concrete semantics of either directly implemented or constructed modules from which other modules can be composed using the composition rules. In the following, we give details on component types and compositional structures for composing voting rules based on the ideas by Diekhoff, Kirsten, and Krämer (2019).

5.1.1 Electoral Modules

The structural foundation of our approach are *electoral modules*, a generalization of voting rules as in Section 3.2. We define electoral modules (see Definition 5.1) so that they act as the principal component type (cf. (i)) within our framework. In contrast to a voting rule, an electoral module does not need to make final decisions for all the alternatives, i.e., partition[1] them (only) into winning and losing alternatives, but can instead defer the decision for some or all of them to other modules. Hence, electoral modules partition the received (possibly empty) set of alternatives $A \subseteq \mathcal{A}$ into *elected*, *rejected* and *deferred* alternatives. In particular, any of those sets, e.g., the set of winning (elected) alternatives, may also be left empty, as long as they collectively still hold all the received alternatives. Just like a voting rule, an electoral module also receives a profile which holds the voters' preferences, which, unlike a voting rule, consider only the (sub-)set of alternatives that the module receives. We take this into account by the following definition of our input domain $\mathcal{D}^{\mathcal{A}}_{\text{mod}}$:

$$\mathcal{D}^{\mathcal{A}}_{\text{mod}} := \{(A, \succsim) \mid A \subseteq \mathcal{A}, \succsim \in \mathcal{L}(A)^+\}$$

$\mathcal{D}^{\mathcal{A}}_{\text{mod}}$ contains all subsets of $\mathcal{A}$ paired with matching profiles. We can hence define electoral modules as follows:

Definition 5.1 (Electoral module) *For eligible alternatives $\mathcal{A}$ and a (sub-)set $A \subseteq \mathcal{A}$, we define an* electoral module *as a function m with*

$$m : \mathcal{D}^{\mathcal{A}}_{\text{mod}} \to \mathcal{P}(\mathcal{A})^3.$$

The function m maps a set of alternatives with a matching profile to the set-triple (e, r, d) of sets of elected (e)*,* rejected (r)*, and* deferred (d) *alternatives such that*

$$(A, \succsim) \in \mathcal{D}^{\mathcal{A}}_{\text{mod}} \Rightarrow (m(A, \succsim) = (e, r, d) \text{ partitions } A).$$

[1] We say that a sequence of sets $s_1, \ldots, s_n$ partitions a set S if and only if S equals the union $\bigcup_{i \in [1,n]} s_i$ over all sets s_i for $i \in [1, n]$ and all their pairwise intersections are empty, i.e., $\forall i \neq j \in [1, n] : s_i \cap s_j = \emptyset$.

In the following, we denote the set of electoral modules by $\mathcal{M}_{\mathcal{A}}$, as well as $m_e(A, \succsim)$, $m_r(A, \succsim)$, and $m_d(A, \succsim)$ for the elected, rejected and deferred alternatives, respectively, of an electoral module m for $(A, \succsim)$.

Moreover, we can easily translate voting rules to electoral modules by returning a triple of empty sets in case the module receives an empty set. Otherwise, we return an empty deferred set, an elected set with exactly the winning alternatives, and a rejected set with the complement of the winning alternatives (we remove the alternatives which are not contained in the received set of alternatives). Note that, as a consequence, social choice properties can also be easily translated in order to conform to electoral modules.

5.1.2 Sequential Composition

Sequential composition (see Definition 5.2) is a compositional structure (cf. (ii)) for composing two electoral modules m, n into a new electoral module $(m \triangleright n)$ such that the second module n only decides on alternatives, which m defers and cannot reduce the set of alternatives that are already elected or rejected by m. In this composition, n receives only m's deferred alternatives $m_d(A, \succsim)$ and a profile $\succsim_{|(m_d(A,\succsim))}$ which only addresses alternatives contained in $m_d(A, \succsim)$.

Definition 5.2 (Sequential composition) *For any set of alternatives $\mathcal{A}$ and a (sub-)set $A \subseteq \mathcal{A}$, electoral modules $m, n \in \mathcal{M}_{\mathcal{A}}$ and input $(A, \succsim) \in \mathcal{D}^{\mathcal{A}}_{\text{mod}}$, we define the sequential composition function $(\triangleright) : \mathcal{M}^2_{\mathcal{A}} \to \mathcal{M}_{\mathcal{A}}$ as*

$$
\begin{aligned}
(m \triangleright n)(A, \succsim) := \quad & (m_e(A, \succsim) \cup n_e(m_d(A, \succsim), \succsim_{|(m_d(A,\succsim))}),\\
& m_r(A, \succsim) \cup n_r(m_d(A, \succsim), \succsim_{|(m_d(A,\succsim))}),\\
& n_d(m_d(A, \succsim), \succsim_{|(m_d(A,\succsim))}))
\end{aligned}
$$

5.1.3 Revision Composition

Mostly for convenience, we define a revision composition (see Definition 5.3) for situations in which we want to revise the alternatives already elected by a prior module, e.g., for enabling sequential composition with a tiebreaking module. For an electoral module m, the revision composition removes m's elected alternatives and attaches them to the previous deferred alternatives, while the rejected alternatives are kept unchanged. Whereas this composition can also be achieved by parallel composition, this dedicated structure turns out to be beneficial in our implementation due to its frequent uses.

Definition 5.3 (Revision composition) *For any set of alternatives $\mathcal{A}$ and a (sub-)set $A \subseteq \mathcal{A}$, electoral module $m \in \mathcal{M}_A$, and input $(A, \succsim) \in \mathcal{D}^{\mathcal{A}}_{\text{mod}}$, we define the revision composition $(\downarrow) : \mathcal{M}_{\mathcal{A}} \to \mathcal{M}_{\mathcal{A}}$ as*

$$(m\downarrow)(A, \succsim) := (\emptyset,\ m_r(A, \succsim),\ m_e(A, \succsim) \cup m_d(A, \succsim)).$$

5.1.4 Parallel Composition

The parallel composition (see Definition 5.6) lets two electoral modules make two independent decisions for the given set of alternatives. Their two decisions are then aggregated by an *aggregator* (see Definition 5.4), which is another component type that combines two set-triples of elected, rejected and deferred alternatives (as well as the set of alternatives) into a single such triple (we define the input domain $\mathcal{D}^{\mathcal{A}}_{\text{agg}}$ accordingly).

Definition 5.4 (Aggregator) *For a set of alternatives $\mathcal{A}$, a (sub-)set $A \subseteq \mathcal{A}$ and input $(A, p_1, p_2) \in \mathcal{D}^{\mathcal{A}}_{\text{agg}}$, an aggregator is a function*

$$\text{agg} : \mathcal{D}^{\mathcal{A}}_{\text{agg}} \to \mathcal{P}(\mathcal{A})^3 \text{ such that } \text{agg}(A, p_1, p_2) \text{ partitions } A.$$

A useful instance of such an aggregator is the maximum aggregator agg_{max}:

Definition 5.5 (Maximum aggregator) *Given two set-triples (e_1, r_1, d_1) and (e_2, r_2, d_2) of elected (e), rejected (r) and deferred (d) alternatives, a maximum aggregator agg_{max} picks, for each alternative a and the sets containing a, the superior one of the two sets with respect to the order $e > d > r$.*

$$\text{agg}_{\text{max}}((e_1, r_1, d_1),\ (e_2, r_2, d_2)) = \\ (e_1 \cup e_2, (r_1 \cup r_2) \setminus (e_1 \cup e_2 \cup d_1 \cup d_2), (d_1 \cup d_2) \setminus (e_1 \cup e_2))$$

Based on the notion of aggregators, we can now define the parallel composition as a function mapping two electoral modules m, n and an aggregator $\text{agg} \in \mathcal{G}_{\mathcal{A}}$ (the set of all aggregators) to a new electoral module $(m \,||_{\text{agg}}\, n)$:

Definition 5.6 (Parallel composition) *For a set of alternatives A and a (sub-)set $A \subseteq \mathcal{A}$, electoral modules m, n, and an aggregator* agg*, we define the parallel composition $(||) : (\mathcal{M}_{\mathcal{A}} \times \mathcal{G}_{\mathcal{A}} \times \mathcal{M}_{\mathcal{A}}) \to \mathcal{M}_{\mathcal{A}}$ as*

$$(m \,||_{\text{agg}}\, n)(A, \succsim) := \text{agg}(A,\ m(A, \succsim),\ n(A, \succsim)).$$

5.1.5 Loop Composition

Based on sequential composition (Section 5.1.2) for electoral modules, we define the more general loop composition for sequential compositions of dynamic length. A loop composition ($m \circlearrowleft_t$) repeatedly composes an electoral module m sequentially with itself until either a fixed point is reached or a *termination condition* t is satisfied. Within our framework, termination conditions, technically another component type, are Boolean predicates on set-triples such that they are suitable for electoral modules. The full definition can be found within the Isabelle/HOL theories provided with this chapter.

5.1.6 A Simple Example

As a simple example, we illustrate the construction of a voting rule using structures from above. Consider the well-known *Baldwin's rule*, which is a voting rule based on sequential elimination (Baldwin, 1926). The rule repeatedly eliminates the alternative with the lowest Borda score (see Section 3.2) until only one alternative remains.

As basic modules (cf. (iv)), we use (a) a module that computes the Borda scores, rejects the alternative with the lowest such score, and defers the rest, as well as (b) a module that attaches all deferred alternatives to the elected set.

Moreover, we choose a termination condition such that the loop of interest stops when the set of (deferred) alternatives has reached size one.

Therefore, Baldwin's rule can be obtained by

1. composing (a) by a loop structure with above-mentioned termination condition, and
2. sequentially composing the loop composition with (b).

Moreover, loop composition can be directly used for many voting rules of a category called *tournament solutions*. Tournament solutions typically consist of multiple rounds, in each comparing a pair of alternatives based on their profile rankings, and the winner of a comparison advances to the next round.

5.2 Compositional Framework

In the following, we describe how we model the concepts defined in Section 5.1 within a modular proof framework for the verified construction of voting rules.

```
type-synonym 'a Preference-Relation = 'a rel
fun is-less-preferred-than ::
  'a ⇒ 'a Preference-Relation ⇒ 'a ⇒ bool (- ≼- - [50, 1000, 51] 50) where
    x ≼r y = ((x, y) ∈ r)

type-synonym 'a Profile = ('a Preference-Relation) list
definition profile :: 'a set ⇒ 'a Profile ⇒ bool where
  profile A p ≡ ∀ i::nat. i < length p ⟶ linear-order-on A (p!i)
fun prefer-count :: 'a Profile ⇒ 'a ⇒ 'a ⇒ nat where
  prefer-count p x y =
     card {i::nat. i < length p ∧ (let r = (p!i) in (y ≼r x))}
fun wins :: 'a ⇒ 'a Profile ⇒ 'a ⇒ bool where
  wins x p y =
    (prefer-count p x y > prefer-count p y x)

type-synonym 'a Result = 'a set * 'a set * 'a set
fun disjoint3 :: 'a Result ⇒ bool where
  disjoint3 (e, r, d) =
    ((e ∩ r = {}) ∧
      (e ∩ d = {}) ∧
      (r ∩ d = {}))
fun set-equals-partition :: 'a set ⇒'a Result ⇒ bool where
  set-equals-partition A (e, r, d) = (e ∪ r ∪ d = A)
fun well-formed :: 'a set ⇒ 'a Result ⇒ bool where
  well-formed A result = (disjoint3 result ∧ set-equals-partition A result)

type-synonym 'a Electoral-Module = 'a set ⇒ 'a Profile ⇒ 'a Result
definition electoral-module :: 'a Electoral-Module ⇒ bool where
  electoral-module m ≡ ∀ A p. finite-profile A p ⟶ well-formed A (m A p)
```

Figure 5.1: Central semantics of electoral modules in Isabelle/HOL.

5.2.1 Verified Construction Framework

We implemented and proved our logical concepts within the interactive theorem prover Isabelle/HOL (Section 2.4), which provides a generic infrastructure for our deductive system that is both human-readable and machine-checked, showing that the deductive conclusions are indeed correct. Thereby, we can define very general theorems to be reused for the construction of various voting rules and sorts of composition. Moreover, the framework allows for easy application and extension within a larger framework for the automatic discovery (Section 5.3) and construction of voting rules, provided that the voting rule of interest can be composed with the given compositional rules and composable modules using the given composition structures and component types.

The definitions and theorems within our framework are mostly self-contained, i.e., for the most part they only rely on basic set theory as well as the theories of finite lists, relations, and order relations for defining the profiles and linear orders used within our notion of profiles and modules as seen in Figure 5.1. Therein, we capture preference relations in a type abbreviation (Line 1), which we use for defining the (within social choice theory) ubiquitous preference semantics in the HOL function *is-less-preferred-than* (Lines 2 to 4). From that basis, we introduce a handy type abbreviation (Line 5) for profiles as lists of preference relations, and we therefrom define profiles (Lines 6 to 7) on alternatives from the theory of order relations, which we use in a number of structures and concepts. Having established a preference semantics and profiles, we can now capture the preference semantics within profiles, e.g., by counting the ballots for which some alternative x is preferred to some other alternative y, as shown in the function in Lines 8 to 10 and the function that makes the comparison in Lines 11 to 13. Moreover, type abbreviations for results of electoral modules (Line 14), i.e., set triples, are introduced, and electoral modules as defined in Section 5.1. We capture the partitioning with the three functions to express the disjointedness of the three sets in an electoral module result (Lines 15 to 19), that their union yields the set of alternatives of the input (Lines 20 to 21), and the conjunction in Lines 22 to 23. Finally, at the end of Figure 5.1, we can essentially define electoral modules on finite profiles and the partitioning of the alternatives (Lines 25 to 26, with *finite-profile* defined appropriately), based on their type abbreviation (Line 24). Besides the theories provided off-the-shelf with the Isabellemain system, our framework does not require any additional theories.

The implementation[1] of our verified construction framework comprises concepts and proofs for 15 composition rules with numerous reusable auxiliary properties and the well-known social choice properties Condorcet consistency (Definition 3.6), monotonicity (Definition 3.8), and homogeneity from the literature, as shown in Figure 5.2.[2]

5.2.2 Verified Construction based on Composition Rules

From devising composition rules and properties as described in the beginning of this section together with the component types and structures as described in Section 5.1, our framework now only requires a small set of basic components in order to construct interesting voting rules for the desired social choice properties which have been defined as properties and included in the rules for composing electoral modules. The power of our approach lies both in the generality of the composition rules and compositional structures such that various voting rules may be constructed for various properties, and in

[1]The full framework is available under https://github.com/VeriVote/verifiedVotingRuleConstruction together with an extensive documentation.

[2]Therein, *times* denotes n-fold concatenation, *elect* the elected set from the *Result* triple, and *lifted* the social choice concept from Definition 3.7.

```
fun condorcet-winner :: 'a set ⇒ 'a Profile ⇒ 'a ⇒ bool where
  condorcet-winner A p w =
    (finite-profile A p ∧ w ∈ A ∧ (∀ x ∈ A − {w} . wins w p x))
definition condorcet-consistency :: 'a Electoral-Module ⇒ bool where
  condorcet-consistency m ≡
   electoral-module m ∧
   (∀ A p w. condorcet-winner A p w ⟶
    (m A p =
     ({e ∈ A. condorcet-winner A p e},
      A − (elect m A p),
      {})))

definition monotonicity :: 'a Electoral-Module ⇒ bool where
  monotonicity m ≡
   electoral-module m ∧
    (∀ A p q w.
      (finite A ∧ w ∈ elect m A p ∧ lifted A p q w) ⟶ w ∈ elect m A q)

definition homogeneity :: 'a Electoral-Module ⇒ bool where
  homogeneity m ≡
   electoral-module m ∧
    (∀ A p n.
     (finite-profile A p ∧ n > 0 ⟶
      (m A p = m A (times n p))))
```

Figure 5.2: Sample social choice properties for our framework in Isabelle/HOL.

the reduction of complexity such that compositions for complex social choice properties can be defined by predominantly local composition rules step by step.

In general, the verified construction using composition rules works as follows: When we want to obtain a voting rule with a set of properties p from some basic components c and d which satisfy sets of properties p_c and p_d respectively, we might make use of a compositional structure X which guarantees that a composed module $m_c X m_d$ satisfies the properties p. Hence, we can get a desired voting rule by instantiating m_c and m_d by c and d respectively, which gives us the induced voting rule f_{cXd}. Note that, when we specify a set of target properties p, any voting rule induced by our framework (if a suitable one can be induced) from a set of basic components and compositional structures, necessarily comes with an Isabelle proof which establishes the validity of p for the induced voting rule. By design, these proofs are short and can in most cases be automatically inferred. Hence, given the soundness of the Isabelle/HOL theorem prover, we obtain a formal proof that the resulting voting rule indeed satisfies the required properties without the need to re-check the obtained rule.

Example. One such example using structures from Section 5.1 and properties defined in this section is that p consists of the property Condorcet consistency, X is the sequential composition, p_c also consists of Condorcet consistency, and p_d is empty. Thus, we have no requirements for properties of any component d, since sequential composition cannot revoke any alternatives that are already elected. If a Condorcet winner exists, this alternative is already elected by the first module, and if not, Condorcet consistency trivially holds. On its own, this composition rule might not be very useful, but may be used in combination with other rules to preserve Condorcet consistency of composed voting rules. A voting rule from the literature which is constructed in such a manner is *Black's rule*. Black's rule is a sequential composition of (a component which induces) a rule that elects the Condorcet winner if there is one, and (a component which induces) the Borda rule.

5.3 Verified Synthesis of Voting Rules

We described our composition framework above with the objective to design flexible composition structures where the major proof load sits within the general composition structures and the modules themselves, such that the specific compositions can be directly inferred from the theorems.

This design decision allows expressing the specific compositions as simple derivations from the properties of the specific modules, which is the answer when queried for a desired property of the full voting rule. With reference to Section 2.1, this structure is directly amenable to the technique of logic programming, when representing the module properties as facts, the composition rules as rule set, and the desired property as the given query. Therefore, this lightweight part of finding compositions for a desired voting rule that has the desired property can be automated with logic programming in order to obtain the desired composition tree – if one exists – automatically, and send these synthesized compositions steps back to Isabelle for doing the full composition proof. We have implemented this with Prolog, where we replaced the default search procedure by iterative deepening to avoid getting lost in infinite branches. This is necessary as the composition of an electoral module may again yield an electoral module, e.g., for sequential composition, but can be easily detected within the new search procedure.

Given the simplicity of the compositions, it then suffices to implement a brute-force procedure that tries complementing each composition step in the proof successively with the most common "general-purpose" Isabelle proof tactics (in our case, this was `simp`, `blast`, `metis`, and `fastforce`) and the proof succeeds within seconds. From this, the Isabelle code generation feature can be used to automatically give us – in addition to the composition proof – a runnable program which we have just verified. With the further

integration of a simple parser for our Isabelle theories, this can be fully automated by producing an intermediate Prolog interpretation of our facts and rules, which can be directly queried in Prolog.

The implementation is based on work by Richter (2021) and is runnable and available for download.[1]

5.4 Evaluation

As a case study for demonstrating the applicability of our approach to existing voting rules and the merits of composition, we constructed the voting rule *sequential majority comparison* (SMC) from the literature (e.g., from Brandt, Conitzer, et al. (2016)), thereby producing a compositional proof that the rule is monotone.

Sequential Majority Comparison (SMC). The voting rule of sequential majority comparison, also known as sequential pairwise majority, is simple enough for understanding, but still complex enough such that it demonstrates interesting properties such as monotonicity. Essentially, SMC fixes some (potentially arbitrary) order on all alternatives and then consecutively performs pairwise majority elections. We start by doing pairwise comparisons of the first and the second alternative, then compare the winner of this pairwise comparison to the third alternative, whose winner is then compared to the fourth alternative, and so on. SMC belongs to a category of voting rules called *tournament solutions*, for which we outline a possible construction pattern in the following.

Verified Construction of Tournament Solution. As indicated in Section 5.1, loop composition appears sensible for *tournament solutions*, as a list of alternatives is processed by multiple rounds, whereof in each, the previously chosen alternative is compared to the next alternative on the list regarding their rankings in the profile, and the winner of a comparison advances to the next round. For the comparison of alternatives, we use any electoral module m which elects one alternative and rejects the rest (for example via plurality voting). In order to limit comparisons to two alternatives, we use the electoral module $pass^2_>$, which defers the two alternatives ranked highest in some fixed order $>$ and rejects the rest. Similarly, $drop^2_>$ rejects these two alternatives and defers the rest. We can now describe a single comparison in our tournament as

$$c = (pass^2_> \triangleright m) \,||_{\mathrm{agg}_{\max}}\, drop^2_>$$

[1]The source code is available under https://github.com/VeriVote/ViRAGe, the tool name stands for "Voting Rule Analysis and Generation."

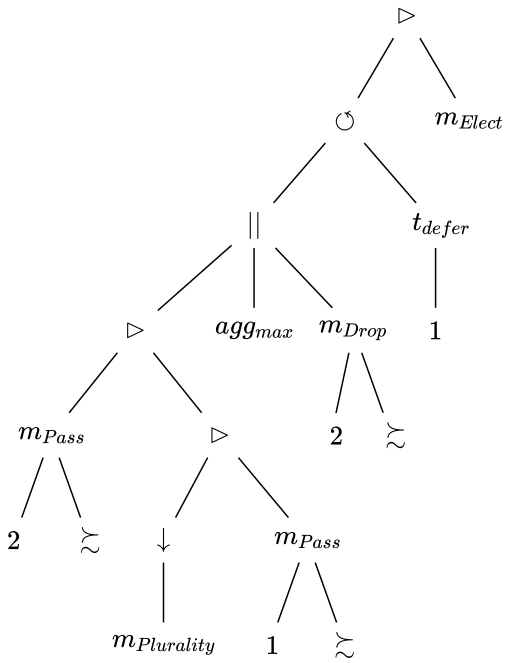


Figure 5.3: Tree representation of the construction for sequential majority comparison

The first part of the parallel composition elects the winner of the current comparison and rejects all other alternatives. The second part defers all alternatives which are not currently being compared and therefore stay in the tournament. The termination condition $t_{|d|=0}$ is satisfied if and only if the set of deferred alternatives passed to it is empty. Then we describe a single round of our tournament as

$$r = (c \circlearrowleft_{t_{|d|=0}}) \downarrow$$

Now, for the case of sequential majority comparison (SMC), we proceed as follows.

Verified Construction of SMC. Every single comparison elects a single alternative to advance to the next round and rejects the other. As long as alternatives are left, the next c compares the next two alternatives. If there ever is only a single alternative left, it advances to the next round automatically. At the end of the round, we need to revise so that we defer all winners to the next round instead of electing them.

Let m_{elect} be the electoral module which elects all alternatives passed to it and $t_{|d|=1}$ the termination condition that is satisfied when exactly one alternative is deferred. We can now define the whole tournament as

$$t = (r \circlearrowleft_{t_{|d|=1}}) \triangleright m_{\text{elect}}.$$

$$\dfrac{\dfrac{\dfrac{\vdots}{m_{\text{elim}} \text{ is non-electing}}}{m_{\text{elim}} \text{ eliminates 1 alternative}}}{m_{\text{elim}} \circlearrowleft_{|d|=1} \text{ defers 1 alternative}} \qquad \dfrac{\dfrac{\vdots}{m_{\text{elect}} \text{ is electing}}}{m_{\text{elim}} \circlearrowleft_{|d|=1} \text{ is non-electing}}}{m_{\text{elim}} \circlearrowleft_{|d|=1} \text{ is defer-lift-invariant}}$$
$$\overline{\qquad\qquad m_{\text{elim}} \circlearrowleft_{|d|=1} \triangleright m_{\text{elect}} \text{ is monotone} \qquad\qquad}$$

Figure 5.4: A simplified excerpt of the top-level monotonicity proof for SMC.

t repeats single rounds as long as there is more than one alternative left, and then elects the single survivor.

Implementing the Construction of SMC in Isabelle/HOL. After having described a general pattern for the verified construction of tournament solutions and sequential majority comparison, we give only structural information on the implemented construction proofs for SMC as the details are rather lengthy, but instead refer the reader to the Isabelle/HOL proofs.

We can construct SMC by combining six different basic components by using all of our composition structures, i.e., the sequential, parallel, loop, and revision structure, and thereby produce a proof that SMC is a monotone voting rule. The composition structure is depicted in Figure 5.3 represented as a tree, where $\gtrsim$ denotes the given profile and the composition structures are represented by their respective icons.

The high-level modular construction can be seen in Figure 5.5, where SMC stands for sequential majority comparison composed of a number of simple components. Each component, the largest of which is an electoral module inducing plurality voting (see Section 3.2), consists of not more than three lines of higher-order logic, and we provided proofs within our Isabelle/HOL framework for easy reuse and modification for similar voting rules.

In Figure 5.4, we illustrate the final deduction rules on the top-level within the monotonicity proof. Therein, m_{elim} stands for the composition before the loop composition, where effectively the loser of the pairwise match is dropped from the original defer-set. Herein, the compositional nature of our framework is visible as the monotonicity property is induced by the properties of the lower tiers.

Moreover, Figure 5.5 shows the simplicity of the abstract proof obligations both that SMC is again an electoral module and satisfies the monotonicity property. Both tasks are proven fully modularly and are hence a direct result of SMC's composition, and is apt for an automated integration within a logic-based synthesis tool. We omit the

```
1  fun smc :: 'a Preference-Relation ⇒ 'a Electoral-Module where
2    smc x A p =
3      ((((((pass-module 2 x) ▷ ((plurality↓) ▷ (pass-module 1 x))) ||↑
4      (drop-module 2 x)) ⟳∃!d) ▷ elect-module) A p)

5  theorem smc-sound:
6    assumes order: linear-order x
7    shows electoral-module (smc x)

8  theorem smc-monotone:
9    assumes linear-order x
10   shows monotonicity (smc x)
```

Figure 5.5: The modular construction of SMC in Isabelle/HOL.

proofs at this point, but they are available for download and can be inspected and replayed for inspection and automatically checked using Isabelle/HOL. The full proof comprises 26 compositions using a set of six basic components within the theorem prover Isabelle/HOL. Besides the proof composition, we made also use of Isabelle's capability to produce a verified program (written in Scala) of the composed SMC rule to directly obtain a runnable program. Using the *ViRAGe* tool described in Section 5.3, we replayed the full process, which makes the presented case study a fully automated synthesis upon the query of an electing monotone voting rule.

5.5 Summary

Within this chapter, we introduced an approach to systematically and automatically synthesize voting rules from compact composable modules to satisfy formal social choice properties. We devised composition rules for a selection of common social choice properties, such as monotonicity or Condorcet consistency, as well as for reusable auxiliary properties. By design, these composition rules give formal guarantees, in the form of an Isabelle proof based on the properties satisfied by the component properties, that a synthesized voting rule fulfills the social choice property of interest as long as its components satisfy specific properties, which we have proved within Isabelle/HOL for the scope of our case study. The described synthesis tool then instruments our formal framework and provides, given the desired properties, a suitable voting rule as a verified and directly runnable Scala program together with the checkable Isabelle proof.

Our approach is applicable to the construction of a wide range of voting rules which use sequential or parallel modular structures, notably voting rules with tiebreakers,

elimination procedures, or tournament structures. This includes well-known rules such as instant-runoff voting, Nanson's method, or sequential majority comparison (SMC).

We synthesized SMC from simple components, which we presume to be reusable for the construction of further rules, and automatically generated a proof that SMC satisfies monotonicity from basic formal proofs for the structures, compositions, and components which we constructed step by step. This case study and all required definitions were implemented and verified with the theorem prover Isabelle/HOL. Finally, our approach can be safely extended with additional modules, compositional structures, and rules, for integration into voting rule design or verification frameworks.

Outlook. So far, composition is realized mostly by transferring sets of deferred alternatives between modules. We also intend to inspect the more involved modular structures already incorporated in some more complicated voting rules in order to achieve a more flexible notion of composition. This, however, also involves making more detailed assumptions on how exactly information is passed between modules, which might come with a loss of generality. Nonetheless, this extension seems to be necessary for voting rules such as Single-Transferable Vote (STV), which are not composable for sensible social choice properties with our strong notion of locality in composition rules. Another sensible extension is the composition of voting rules based on *distance rationalization* that is very flexible, as almost every voting rule can be constructed this way with the caveat that the composition structures would be less imperative than the structures described in this chapter (Elkind, Faliszewski, and Slinko, 2015; Steinriede, 2021).

6

Efficient Verification of Reliable Tallying Procedures

> The real value of tests is not that they detect bugs in the code, but that they detect inadequacy in the methods, concentration and skills of those who design and produce the code.
>
> Charles Antony Richard Hoare, *How Did Software Get So Reliable Without Proof?*, 1996

Having seen how to construct and synthesize voting rules from given requirements in the previous chapter, this chapter considers the case where we already have a designed or even implemented version of a voting rule or multiple thereof. However, it is not always easy to decide right away on the desired requirements for a voting rule, or the environment of the envisioned application might change, given that the tradeoff between any such required properties is inherently difficult and error-prone. The experiences presented in this chapter document that errors in voting rules are easy to make, and formal methods greatly enhance the chances of finding such errors. Furthermore, we document that there are formal techniques which provide proof that a voting rule – and its algorithmic implementation – meets a given property.

Within this chapter, we are particularly interested in means to justify, compare, or adapt an already existing voting rule in a comprehensible and efficient manner. Concretely, we investigate the power of SBMC techniques (Section 2.2) for verifying properties of voting rules, demonstrated on simple examples, and for arguing for or against a given voting rule when compared to another voting rule.

We start by giving insights into the logical models and formalizations that we used, and also present formalization techniques specially tailored to axiomatic properties of voting rules. Then, we describe and compare techniques used for an efficient application of formal program verification on voting rules.

The content of this chapter has been previously published by Kirsten and Cailloux (2018), Beckert, Bormer, Goré, et al. (2017), and Beckert, Bormer, Kirsten, et al. (2016).

6.1 Functional and Relational Properties

We consider voting rules as instances of preference aggregation problems, where the individual preferences of voters are combined to produce an election result. The input for the voting rule is modeled as a finite sequence of ballots, with one ballot for each voter. For this section and the next one, we give general classifications of social choice properties, which we formalize based on a notion of voting rules in Definition 6.1 that is more general than the ranking-based notion in Definition 3.1. Thereby, the classifications are applicable to a wider range of voting rules and profiles, e.g., in settings where voters assign scores to each alternative.

Definition 6.1 (General voting rule, profile) *Given a set $\mathcal{B}$ of possible ballots and a set W of possible election results, a* voting rule *is a total function $f : \mathcal{B}^* \to W$, assigning an election result to each profile, where a* profile *$B \in \mathcal{B}^*$ is a finite sequence of ballots.*

The special election result $\bot$ may be included in W to denote that there is no "valid" result (e.g., in case of a tie).

An individual pair $(B, W) \in (\mathcal{B}^* \times W)$ consisting of a profile and an election result will be called an evaluation. *The set of all evaluations is $\mathcal{E} = \mathcal{B}^* \times W$.*

The concrete structure of ballots $b \in \mathcal{B}$, how ballots encode voters' preferences, and the structure of possible election results in W depend on the investigated voting rule. In examples throughout this chapter and in the case study in Section 6.4, we use preferential voting rules with single winners:[1]

Definition 6.2 (Preferential voting rule) *Let $\mathcal{A}$ be a finite set of* alternatives. *A voting rule is called* preferential *if the possible ballots $\succsim_b \in \mathcal{B}$ are linear orders on $\mathcal{A}$ and the possible results are single alternatives or a tie, i.e., $W = \mathcal{A} \cup \{\bot\}$.*

We distinguish between *functional* and *relational* properties. Functional properties, such as the majority criterion, refer to single election results while relational properties, such

[1] Definition 6.2 is very similar to Definition 3.1, except that the result is either a single alternative or otherwise considered invalid, i.e., we do not distinguish different tie-situations, but abstract them under the value $\bot$.

as anonymity, compare two (or more) results. In the literature, functional and relational properties are also called *intra-* and *interprofile* properties, as defined by Fishburn (1973).

Definition 6.3 (Functional property) *Given a set $\mathcal{B}$ of possible ballots and a set W of possible results, a* functional property *F for voting rules is a set of evaluations, i.e., $F \subseteq \mathcal{E} = (\mathcal{B}^* \times W)$ is a relation between profiles and results.*

A voting rule $f : \mathcal{B}^* \to W$ satisfies a functional property *F if and only if $f \subseteq F$, i.e., all evaluations of f are elements of F. Intuitively, a functional property F is the set of those evaluations that a voting rule may contain if it is to have that property.*

Example 6.4 (Majority criterion, majority winner) *Given a profile $p \in \mathcal{B}^*$, a majority winner for $p = \langle \succsim_1, \ldots, \succsim_n \rangle$ is an alternative $a \in A$ that is preferred over all other alternatives in more than half of the ballots:*

$$|\{\succsim_i \in p \mid a \succsim_i a' \text{ for all } a' \in A, a' \neq a\}| > \tfrac{n}{2} \ .$$

A voting rule satisfies the majority criterion if and only if, for all profiles p, either the majority winner for p is elected or there is no majority winner for p. This criterion is formalized by the functional property

$$\mathit{Maj} \ = \ \{(p, a) \mid \text{if } a' \text{ is a majority winner for } p, \text{ then } a' = a\} \ .$$

Definition 6.5 (Relational property) *Given a set $\mathcal{B}$ of possible ballots and a set W of possible results, a* relational property *R for voting rules is a set of pairs of evaluations, i.e.,*

$$R \subseteq \mathcal{E} \times \mathcal{E} = (\mathcal{B}^* \times W) \times (\mathcal{B}^* \times W) \ .$$

A voting rule $f : \mathcal{B}^* \to W$ satisfies a relational property *R if and only if, for all evaluations $e, e' \in f$, the pair (e, e') is in R.*

Intuitively, a relational property R consists of those pairs of evaluations that – by definition of that property – are allowed to "coexist" in a voting rule.

Example 6.6 (Monotonicity criterion) *For the monotonicity criterion, we need to compare profiles that are identical up to one ballot. By $b^{\uparrow a} \subset \mathcal{B}$, we denote the set of all ballots that are identical to b, except that now $a \in A$ is given a higher rank, i.e., a is* lifted.

The relational property Mono *of monotonicity is as follows:*

$$\mathit{Mono} \ = \ (\mathcal{E} \times \mathcal{E}) \ \setminus \ \{((p, w), (p', w')) \mid \begin{array}{l}\text{there is an alternative with } a \in w, a \notin w', \\ \text{and } p' \text{ results from } p \text{ by replacing} \\ \text{a single ballot } b \in p \text{ by a ballot } b' \in b^{\uparrow a}\}\end{array}$$

That is, Mono contains all *pairs of evaluations* except *those where a winning alternative is given higher preference in one of the ballots (denoted by b') which results in the alternative a no longer being elected.*

A functional property consists of single evaluations, namely those evaluations that are considered "good" by the property. A voting rule is judged against the functional property for every evaluation separately. In contrast, a relational property is a relation between two evaluations of the voting rule. Satisfaction is hence judged by considering each of its evaluations in the context of the other evaluations. Thus, the concept of relational properties is stronger and more expressive. In fact, every functional property can also be represented as a relational property.

The classes of functional and relational properties do not cover all interesting properties of voting rules, but only those that can be checked by looking at one (functional) or two (relational) evaluations at a time. However, there are properties that require a comparison of three or more evaluations.

Example 6.7 (Consistency criterion) *A voting rule satisfies the consistency criterion if, for any three profiles p, p_1, p_2 such that p is the concatenation of p_1 and p_2:*

$$\text{if } f(p_1) = f(p_2), \text{ then } f(p) = f(p_1) = f(p_2).$$

Properties such as consistency,[1] which can (only) be defined by comparing three evaluations are called 3-properties. This concept can be extended to generalized k-properties for $k \in \mathbb{N}$, which does – however – still not cover all properties. For example, the surjectivity[2] property, which requires that for each possible election result there is a profile leading to that result, is a rather simple property that is not a k-property for any k. Surjectivity is an *existential* property, requiring the existence of (combinations of) certain evaluations, while k-properties are universal in nature, requiring all k-tuples of evaluations are "good" in some sense.

6.2 Exploitation of Symmetry Properties

An important kind of relational properties are those expressing that, if two profiles are symmetric (or in some way similar) to each other, then they lead to symmetric (similar) election results. Many fairness criteria are of this type.

In practice, the amount of possible ballots is very large and the amount of possible profiles even larger. Correspondingly, there is a huge amount of possible execution paths through

[1] Consistency is sometimes also referred to as *reinforcement* or *convexity*.
[2] Sometimes, surjectivity is also referred to as (strict) non-imposition property.

implementations of voting rules. Exploiting symmetries is an important technique so that testing or formal verification attempts gain in feasibility.

The idea is to only prove that a voting rule f satisfies a functional property F for a small subset $X \subseteq \mathcal{B}^*$ of the possible profiles, i.e., $(x, f(x)) \in F$ for all $x \in X$, and to then make use of the symmetry property in order to conclude that the same holds for all profiles $p \in \mathcal{B}^*$, i.e., f has the property F in general. This, of course, is only useful if either both the subset X is much smaller than $\mathcal{B}^*$ and if it is easy to prove that f is symmetric with respect to a symmetry relation S for which X are the representatives, or if we can assume an existing proof since the symmetry is an interesting property in its own right (anonymity, neutrality, monotonicity, etc.). In the specification used for verification, the restriction to the set X is achieved by a first-order logic predicate ψ, called a *symmetry-breaking predicate* (SBP), a term originating from the field of constraint satisfaction (Crawford et al., 1996). The formula $\psi(P)$ has a free variable P, and $X_\psi \subseteq \mathcal{B}$ is the set of profiles that satisfy $\psi(P)$.

6.2.1 Symmetry Properties

An important kind of relational properties includes those expressing that, if two profiles are symmetric (or in some other way similar) to each other, then they lead to symmetric (similar) election results. Many criteria for voting rules are of this type.

Definition 6.8 (Symmetry property) *A* symmetry property *is a relational property*

$$S_{\sim,\approx} \;=\; \{((B, a), (B', a')) \mid B \sim B' \text{ implies } a \approx a'\},$$

where $\sim \subseteq \mathcal{B}^* \times \mathcal{B}^*$ and $\approx \subseteq W \times W$ are binary relations on profiles respectively election results.

A voting rule satisfying $S_{\sim,\approx}$ (Definition 6.5) is called symmetric *with respect to $S_{\sim,\approx}$.*

Thus, a voting rule f is called *symmetric* with respect to $S_{\sim,\approx}$ if two $\sim$-related profiles yield $\approx$-related results. Note that, according to our definition of symmetry, it is not a requirement for $\sim$ or $\approx$ to be symmetric relations themselves.

Example 6.9 (Anonymity criterion (Fishburn, 1973)) *One elementary symmetry property, which most voting rules satisfy, is* anonymity*, where $\sim_{\text{anon}}$ is defined by*

$$B \sim_{\text{anon}} B' \text{ iff } B' \text{ is a permutation of } B$$

and $\approx_{\text{anon}}$ is the equality relation on W. The symmetry property $S_{\sim_{\text{anon}},\approx_{\text{anon}}}$ expresses that changing the order of the ballots – corresponding to changing which voter casts which vote – does not affect the election result. In this example, both $\sim_{\text{anon}}$ and $\approx_{\text{anon}}$ are symmetric.

It is often useful to consider families of symmetries that are parameterized by elements of some set P (which may, for example, contain permutations):

Definition 6.10 (Family of symmetry properties) *Given a parameter set P, a* family of symmetry properties *is a set $\{S_{\sim_p,\approx_p} \mid p \in P\}$ of symmetry properties that are induced by relational properties $\sim_p, \approx_p$ for each $p \in P$.*

Example 6.11 (Neutrality criterion (Fishburn, 1973)) *Again, we consider preferential voting rules (Definition 6.2). The neutrality criterion expresses that, if the alternatives are permuted in a profile, then the voting rule permutes the alternatives in the same way in the election result.*

Neutrality can be formalized as a family of symmetry properties using the parameter set $P = \{\pi \mid \pi$ is a permutation on $A\}$ and defining:

$$B \sim_\pi B' \quad \text{iff} \quad B' = \pi(B) \qquad \text{and} \qquad a \approx_\pi a' \quad \text{iff} \quad a' = \pi(a) \ ,$$

(where the application of π is extended to ballots and profiles in the obvious way). Thus,

$$S_{\sim_\pi,\approx_\pi} \ = \ \{((B,a),(B',a')) \mid B' = \pi(B) \text{ implies } a' = \pi(a)\} \ .$$

A voting rule satisfies the neutrality criterion if and only if it satisfies $S_{\sim_\pi,\approx_\pi}$ for all $\pi \in P$.

Example 6.12 (Monotonicity symmetry property) *The monotonicity property Mono in Example 6.6 also corresponds to a family of symmetry properties, using the parameter set $P = A$ (the alternatives are the parameters) and defining $\sim_{\uparrow p}, \approx_{\uparrow p}$ by:*

$$\begin{aligned} B \sim_{\uparrow p} B' \quad &\text{iff} \quad B' \text{ results from } B \text{ by replacing a single ballot } b \in B \text{ by a ballot } b' \in b^{\uparrow p} \\ a \approx_{\uparrow p} a' \quad &\text{iff} \quad a = p \text{ implies } a' = p. \end{aligned}$$

A voting rule satisfies property Mono if and only if it satisfies $S_{\sim_{\uparrow p},\approx_{\uparrow p}}$ for all $p \in A$, i.e., $Mono = \bigcap_{p \in A} S_{\sim_{\uparrow p},\approx_{\uparrow p}}$.

Note that the relation $\sim_{\uparrow p}$ is not an equivalence relation for monotonicity.

This formalization using a parametric property family is necessary for *Mono*, since it cannot be described as a single symmetry property. In fact, every relational property can be phrased as a family of symmetry predicates for some parameter set.

6.2.2 Symmetry Exploitation

In addition to establishing $(x, f(x)) \in F$ for all $x \in X$, we also have to establish (1) that f has symmetry property S, i.e., it produces symmetric outputs for symmetric inputs, (2) all elements in $\mathcal{B}^*$ are represented by (i.e., are symmetric to) at least one element in $\mathcal{B}^*$ which satisfies ψ, and (3) for any evaluation (p, w) satisfying property F, all evaluations (p', w') symmetric to (p, w) also satisfy F. Note, that only (1) needs to be proven for the specific voting rule f, while (2) and (3) only depend on F, S, and X. Propositions (2) and (3) can hence be verified either via a manual proof, or using an automated theorem prover that can deal with first-order logic and set theory (including transitive closure). The proof for $(x, f(x)) \in F$ can be done using program verification techniques, using ψ as an assumption in the proof.

Definition 6.13 (Spanning set) *Let $\sim \subseteq \mathcal{B}^* \times \mathcal{B}^*$ be a binary relation on profiles. A set $X \subset \mathcal{B}^*$ of profiles is a* spanning set with respect to $\sim$ *if and only if, for every profile $B \in \mathcal{B}^*$, there is a profile $x \in X$ with $x \sim^* B$, where $\sim^*$ is the transitive, reflexive closure of $\sim$.*

Definition 6.14 (Symmetry resilience) *Let $F \in \mathcal{F}$ be a functional property, and let $S_{\sim,\approx} \subseteq \mathcal{R}$ be a symmetry property. Then, F is called* symmetry-resilient with respect to $S_{\sim,\approx}$ *if and only if, for all $B, B' \in \mathcal{B}^*$ and $W, W' \in W$, then*

$$(B, W) \in F, B \sim B', W \approx W' \qquad \textit{implies} \qquad (B', W') \in F.$$

The following theorem formalizes the approach of exploiting symmetry properties to help with proving functional properties.

Theorem 6.15 (Symmetry breaking) *Let $F \in \mathcal{F}$ be a functional property and $\{S_{\sim_p,\approx_p}\}$ a family of symmetry properties with parameter set P; further, let $X \subset \mathcal{B}^*$ be a set of profiles.*

Then, every voting rule $f : \mathcal{B}^* \to W$ such that

$$f \text{ is symmetrysymmetric (Definition 6.8) with respect to } S_{\sim_p,\approx_p} \text{ for all } p \in P \tag{6.1}$$

$$(x, f(x)) \in F \text{ for all } x \in X \tag{6.2}$$

$$X \text{ is spanning (Definition 6.13) with respect to the relation's union } \bigcup_{p \in P} \sim_p \tag{6.3}$$

$$F \text{ is symmetry-resilient (Definition 6.14) with respect to } S_{\sim_p,\approx_p} \text{ for all } p \in P \tag{6.4}$$

satisfies the property F, i.e., $f \subseteq F$.

Proof Let $B \in \mathcal{B}^*$ be an arbitrary profile; we have to show that $(B, f(B)) \in F$. Because of (6.3), there exist $B_0 \sim_{p_0} B_1 \sim_{p_1} \cdots \sim_{p_{n-1}} B_n = B$ with $B_0 \in X$. It is, thus, sufficient to show

that $(B_k, f(B_k)) \in F$ for all k, which we do by induction on k. The base case $(B_0, f(B_0)) \in F$ follows from (6.2) as $B_0 \in X$. For the step case, the induction hypothesis is $(B_k, f(B_k)) \in F$. As $B_k \sim_{p_k} B_{k+1}$, we have also $f(B_k) \approx_{p_k} f(B_{k+1})$ by (6.1). From that and the hypothesis, we can derive $(B_{k+1}, f(B_{k+1})) \in F$ using (6.4). ∎

The following corollary is a version of Theorem 6.15, in which all definitions are expanded. We include it to show in one place the four proof obligations that arise for verifying that the voting rule f satisfies the functional property F using the symmetry property S.

Corollary 6.16 (Symmetry breaking (expanded)) *Let $F \in \mathcal{F}$ be a functional property and $\{S_{\sim_p, \approx_p}\}$ a family of symmetry properties with parameter set P; further, let $X \subseteq \mathcal{B}^*$ be a set of profiles.*

Then, every voting rule $f : \mathcal{B}^* \to W$ such that

$$\forall B, B' \in \mathcal{B}^*, p \in P : \quad B \sim_p B' \implies f(B) \approx_p f(B') \tag{6.1'}$$

$$\forall x \in X : \quad (x, f(x)) \in F \tag{6.2'}$$

$$\forall B \in \mathcal{B}^* : \exists x \in X : \quad x \sim^* B \tag{6.3'}$$

$$\forall (B, W), (B', W') \in \mathcal{E}, p \in P : \quad (B, W) \in F \land B \sim_p B' \land W \approx_p W' \Rightarrow (B', W') \in F \tag{6.4'}$$

satisfies the property F, i.e., $f \subseteq F$.

In (6.3'), $\sim^*$ denotes the transitive, reflexive closure of $\bigcup_{p \in P} \sim_p$.

6.3 Efficient Relational Verification via Program Weaving

Relational properties (Definition 6.5) relate the behavior of a voting rule for two independent input (profiles). For verification, two runs of the same program α, implementing the voting rule, need to be analyzed and their results compared. A common technique, called *self-composition* (Darvas, Hähnle, and Sands, 2005; Barthe, Crespo, and Kunz, 2011), for proving a relational property for a program α is to show a functional property for the concatenation "$\alpha_1 ; \alpha_2$", combining the behavior of two variants α_1 and α_2 of α that are identical up to variable names, hence operating on disjoint variable sets, and storing the outputs in disjoint variable sets as well. Based on Hoare logic (Hoare, 1969), we then verify a relational property R by running "$\alpha_1 ; \alpha_2$" with (symbolic) inputs p_1, p_2 with the results w_1, w_2, and proving $((p_1, w_1), (p_2, w_2)) \in R$.

Formal verification of relational properties using self-composition is challenging in general, since it requires static analysis of two independent program runs; the exploration space that needs to be analyzed is potentially exponentially larger than the exploration

space for analyzing a single program run. Moreover, for this type of relational verification, sufficiently strong program specifications (in particular, loop invariants and postconditions) are required to prove non-trivial properties.

Another way to handle relational verification, which improves on self-composition, is to weave the two variants into a single combined program. Since α_1 and α_2 have disjoint variable sets, reordering statements cannot have an effect on the result as long as the execution order of statements is preserved. Details about the possibilities of flexibly weaving programs can be found in the work by Felsing et al. (2014) and Barthe, Crespo, and Kunz (2011). Consider for instance the program "`while (cond) { body }`" consisting of a single while-loop. It is easy to see that, instead of concatenating two variants of this code (one with $cond_1$ / $body_1$ and one with $cond_2$ / $body_2$), one can use the following single-loop program:

while ($cond_1$ || $cond_2$) { **if** ($cond_1$) {$body_1$} **if** ($cond_2$) {$body_2$} }

This weaved program does not require separate loop invariants for the loooploops in α_1 and α_2 but only a single so-called *coupling invariant* for the weaved loop that sets variables $\bar{x}_1$ and $\bar{x}_2$ into relation. In many cases, the coupling invariant is significantly simpler than the (functional) loop invariants. As long as the two loop executions behave similarly, it is easier to express how the two states are related after each step than to specify what it is that the loops actually compute.

6.4 Relational Verification of Voting Rules

Below, we report on experiences with relational verification and symmetry breaking techniques (see Sections 2.2 and 6.3). For our case study, we used the automated software (bounded) model checker CBMC (Clarke, Kroening, and Lerda, 2004), which takes C/C++ programs as input that are annotated with specifications in the form of assertions and assumptions. We performed our computations using CBMC's SAT backend based on MiniSat (Eén and Sörensson, 2003) in combination with an efficient bit-vector refinement procedure (Bryant et al., 2007).

Relational Verification Using CBMC. As explained in Section 6.3, relational verification with weaved programs and coupling invariants is often more efficient than just composing two variants of the (same) program. We evaluate the impact of coupling invariants on performance and feasibility, using as an example the verification of simple first-past-the-post plurality voting with respect to the anonymity property. For plurality voting, anonymity can be written as follows:

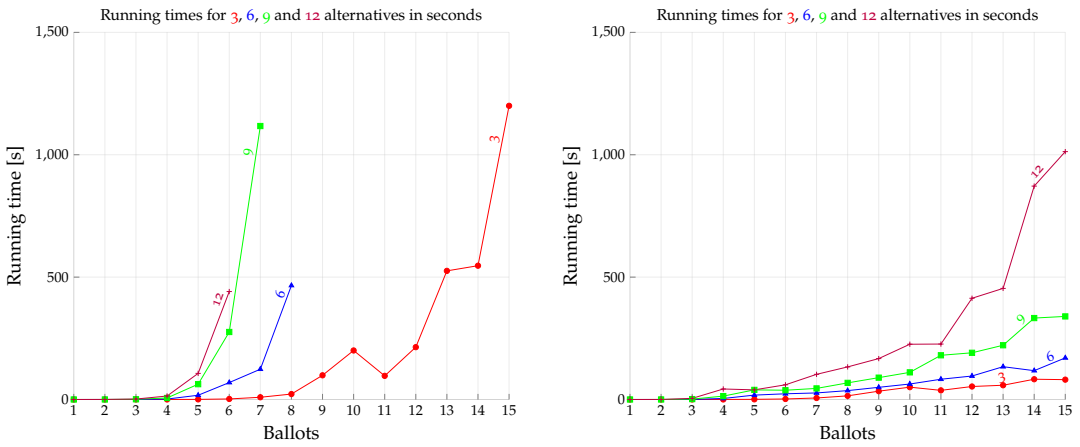
```
1  void anonymity(int p1[N], int p2[N]) {
2      for (int i = 1; i <= N; i++) {
3          assume (0 < p1[i] ≤ M ∧ 0 <p2[i] ≤ M) }
4      int b1, int b2; assume (0 < b1 ≤ N ∧ 0 < b2 ≤ N ∧ b1 < b2);
5      assume (p1[b1] == p2[b2] ∧ p2[b1] == p1[b2]);
6      for (int i = 1; i <= N; i++) {
7          if (i != b1 && i != b2) assume (p1[i] == p2[i]); }
8      int w1 = plurality(p1);
9      int w2 = plurality(p2);
10     assert (w1 == w2);
11 }
```

Listing 6.1: Anonymity property as a C program

$$\begin{aligned}\phi_{Anon}(P_1, W_1, P_2, W_2) \;=\; \forall b_1, b_2 (\;\; (\;\; & \forall i (0 < i \leq n \rightarrow (0 < P_1[i] \leq m \wedge 0 < P_2[i] \leq m)) \\ & \wedge\; 0 < b_1 < b_2 \leq n \\ & \wedge\; P_1[b_1] = P_2[b_2] \wedge P_2[b_1] = P_1[b_2] \wedge \\ & \forall i ((0 < i \leq n \wedge i \neq b_1 \wedge i \neq b_2) \rightarrow P_1[i] = P_2[i]) \\ ) \rightarrow & \; W_1 = W_2)\end{aligned}$$

Anonymity is a relational property (see Section 6.1). It is formalized as a first-order logic formula $\phi_{Anon}(P_1, W_1, P_2, W_2)$, using four free variables denoting two profiles and two election results, respectively. Since plurality voting is a single-choice voting rule (it is not preferential), we assume the profiles to be one-dimensional arrays, i.e., the ith ballot $p[i]$ of profile p equals the ith voter's single choice and is not itself an array. Moreover, we assume that, in case of a tie, no alternative is elected and that this is indicated by the election result of $w = 0$.

Listing 6.1 shows the corresponding CBMC specification, expressing that the voting rule implemented in the C function `plurality` satisfies the anonymity property. Lines 2 and 3 express the assumption that the profiles are well-formed. The universal quantification of variable `i` is expressed using a **`for`**-loop. The variables b_1, b_2 introduced by the assumption in Line 4 are implicitly universally quantified, as CBMC carries out the proof for all values satisfying the assumptions. The profiles p_1, p_2 are assumed to only differ in ballots of voters b_1, b_2, in that the ballots of these two voters are exchanged. This is expressed in Lines 5 to 7. The function `plurality` is invoked in Lines 8 and 9 to compute the election result for the two profiles p_1, p_2. Finally, Line 10 makes the assertion that the two election results are identical. CBMC will prove that this assertion holds for all inputs (a) whose size is within a given bound and that (b) satisfy the assumptions from Lines 2 to 7.



(a) Without coupling invariants

(b) Weaved program with coupling invariants

Figure 6.1: Verification of the anonymity property for plurality voting

We used CBMC to verify the anonymity property for plurality voting using (a) simple composition of two variants without coupling invariants and (b) weaved programs with coupling invariants for the loops (the implementation of plurality voting has two loops, one counting the ballots for each alternative and one for finding the alternative with the maximum amount of votes).

Figure 6.1 shows the running times (in seconds) for between 1 and 12 alternatives and 1 to 15 ballots. For the missing data points, the running times exceed our predefined timeout of 30 minutes. The results show that the verification without weaving and coupling invariants becomes infeasible for rather small bounds. Verification with coupling invariants fares considerably better; the timeout, here, is finally reached for about 10 alternatives and 25 ballots (in contrast to only 10 alternatives and 7 ballots without coupling invariants).

Symmetry Breaking. We continue the case study, now with the goal to verify the majority criterion (Example 6.4) for plurality voting.

Using CBMC in a straightforward manner, verification is possible for small bounds on the amounts of voters and alternatives, but becomes infeasible for higher amounts. For example, a timeout of 30 minutes is reached with 5 alternatives and 45 voters, respectively with 10 alternatives and 20 ballots. Considering the small-scope hypothesis and the simple structure of plurality voting, we deem these bounds high enough. The running times (in seconds) for 10 alternatives are shown in the second column of Table 6.1 ('t/o' indicates timeout). The full data can be seen in the work by Beckert, Bormer, Kirsten, et al. (2016).

Ballots	Without Symm. Red.	With Symm. Red.
5	1,2	0,2
10	41,7	0,9
15	84,3	3,8
20	t/o	6,9
50	t/o	194,2
80	t/o	747,9
85	t/o	855,4
90	t/o	1 369,6
95	t/o	t/o

Table 6.1: Verification of the majority property for plurality voting for 10 alternatives (running times in seconds)

Using symmetry breaking, however, the efficiency of the verification can be considerably increased – and, thus also, the reachable bounds. Assuming anonymity, which is a symmetry property, by applying the symmetry-breaking predicate

$$\forall i(0 < i < n) \rightarrow P[i-1] \leq P[i],$$

the situation improves dramatically. This predicate requires the ballots to be sorted according to which alternatives they prefer. Intuitively, this is a valid assumption, as anonymity allows to re-order the ballots.

The much lower running times are shown in the right column of Table 6.1. Experiments show that handling more than 100 ballots for 10 alternatives becomes feasible, when adding predicates for further symmetry properties.

6.5 Efficient Generation of Counterexamples

In this section, we further investigate on using SBMC – which we already applied above – for determining whether a rule satisfies an axiom within the specified bounds. Our first interest was and still is to detect automatically which axioms a voting rule satisfies among a given set of axioms.

It is, moreover, especially interesting to obtain short and readable proofs that a rule does not satisfy some axiom, which is our main objective in this section: our automatic prover outputs profiles, preferably small or satisfying some other property representing "simplicity" for a human eye, that exhibit a failure of a rule to satisfy an axiom. Such a short proof is interesting even if it is already known in the literature that the rule does not

satisfy the axiom, as short proofs permit to help get an intuitive understanding of voting rules. Furthermore, it can be used to talk about voting rules with non-expert users.

We illustrate it by applying the approach to the case of Borda: we consider a set of four axioms that characterize the Borda rule, and given some rules different from the Borda rule, we find automatically which axioms from that set each rule fails to satisfy. Our illustration in this case permits to obtain a tool that "argues in favor" of Borda (Cailloux and Endriss, 2016) (assuming the axioms characterizing Borda of our choice are considered desirable properties of any voting rule): given any rule that is not the Borda rule, the prover finds a concrete profile which illustrates that the rule fails to comply with some of those axioms.

We start by two further properties, of which we know that they are satisfied by Borda, Pareto dominance (or simply *Pareto*) and reinforcement.

The property Pareto dominance Dom is an example of a functional property.

Definition 6.17 (Pareto dominance) *Pareto dominance (Dom) forbids alternatives that are Pareto-dominated from winning. An alternative $a \in \mathcal{A}$ is Pareto-dominated in a profile $(\succsim_i)_{i\in N}$ if and only if some alternative $a' \in \mathcal{A}$ is unanimously preferred to a in $(\succsim_i)_{i\in N}$: $\forall i \in N, a' \succsim_i a$. The property mandates that the rule selects winning alternatives among U_R, denoting the alternatives that are not Pareto-dominated in R: $\forall R \in \bigcup_{N\subseteq\mathcal{N}} \mathcal{L}(\mathscr{A})^N$, $S_{Dom}(R) = \mathscr{P}^*(U_R)$.*

The property reinforcement Reinf exemplifies a 3-relational property.

Definition 6.18 (Reinforcement) *Reinforcement (Reinf) requires that elections which unite disjoint groups of voters elect the alternatives chosen by both groups, if some such alternatives exist: for each $(\succsim_i)_{i\in N_1}, (\succsim_i)_{i\in N_2}$ with $N_1 \cap N_2 = \emptyset$ and $A \neq \emptyset$, defining A as $f((\succsim_i)_{i\in N_1}) \cap f((\succsim_i)_{i\in N_2})$, reinforcement imposes that $f((\succsim_i)_{i\in N_1\cup N_2}) = A$.*

For the following experiments, we use CBMC 5.8 (Clarke, Kroening, and Lerda, 2004), an implementation of the SBMC approach for the C language, with the built-in solver based on the SAT solver MiniSat 2.2.0 (Eén and Sörensson, 2003). All experiments[1] are performed on an Intel® Core™ i5-6500 CPU at 3.20 GHz with 4 cores and 16 GB of RAM.

A Simple Example

Listing 6.2 illustrates how we specify Dom, the functional property *Pareto dominance*. The function is annotated with specifications in the form of assumptions (expressed by the

[1]The implementations within this section and many more together with a convenient bash script that presents readable counterexamples are available under github.com/mi-ki/voting-rule-argumentation/.

```
1 void dominance(uint prof[N][M], uint win[M]) {
2   uint bad = nondet_uint(), good = nondet_uint();
3    __CPROVER_assume (0 ≤ bad < M);
4    __CPROVER_assume (0 ≤ good < M);
5    __CPROVER_assume (bad ≠ good);
6    uint prefergtob[N];
7    for (uint i = 0; i < N; i++) {
8       uint rankg = M, rankb = M;
9       for (uint r = 0; r < M; r++) {
10         if (prof[i][r] == good) rankg = r;
11         if (prof[i][r] == bad) rankb = r;
12      }
13      prefergtob[i] = (rankg < rankb) ? 1 : 0;
14   }
15   for (uint i = 0; i < N; i++)
16      __CPROVER_assume (prefergtob[i]);
17   __CPROVER_assume (win[bad]);
18 }
```

Listing 6.2: Pareto-dominance specification for CBMC.

function `__CPROVER_assume`), that represent the statements composing C^{ant}, as well as operations to model nondeterministic choice (indicated by the prefix `nondet_`) that represent what we called nondeterministic parameters in Section 2.2. These nondeterministic-choice operators can be used anywhere inside the analyzed program and are also translated to the formula which is passed to the solver. The solver then searches for instantiations of these variables which lead to a violation of C^{cons}.

In the displayed function, the constants `N` and `M` denote the amounts of voters and alternatives, respectively. The voters and alternatives are represented by integers from 0 to `N` − 1 and from 0 to `M` − 1. The function accepts as input a profile `prof`, modeled as a two-dimensional array: `prof[i][r]` is the alternative that voter `i` associates to rank `r`, where `0` is the best, and `M` − 1 the worst rank. The function also accepts as input a set of winning alternatives, modeled as an array `win` with one entry per alternative: `win[a]` equals `1` if the alternative `a` belongs to the set of winning alternatives, `0` otherwise. The nondeterministic parameters are `prof` (whose declaration is not shown here) and the two alternatives `bad` and `good` (Line 2). Lines 6 to 14 initialize the array `prefergtob` so that `prefergtob[i]` holds the value `1` if voter `i` prefers `good` to `bad`, `0` otherwise.

The function thus indicates to the solver that it must find two alternatives `bad` and `good`, thus integers in the suitable range (Lines 3 and 4) and different from each other (Line 5), such that every voter prefers `good` to `bad` (Lines 15 and 16), and yet the alternative `bad` is a winner of the election (Line 17). Any run which satisfies these specified statements is a

```
1  int main(int argc, char *argv[]) {
2     uint prof[N][M], uint win[M];
3     for (uint i = 0; i < N; i++) {
4        uint used[M];
5        for (uint a = 0; a < M; a++) used[a] = 0;
6        for (uint r = 0; r < M; r++) {
7           a = nondet_uint();
8           __CPROVER_assume (0 ≤ a < M);
9           __CPROVER_assume (!used[a]);
10          prof[i][r] = a;
11          used[a] = 1;
12       }
13    }
14    win = f(prof, N);
15    dominance(prof, win);
16    assert (0);
17    return 0;
18 }
```

Listing 6.3: Setup for CBMC.

valid counterexample which would prove that the profile and winners given as input violate Pareto dominance.

We pursue the example with the setup given in Listing 6.3. Therein, we initialize a profile with symbolic nondeterministic values (Line 7) and restrict their ranges such that they are valid alternatives (Line 8). Furthermore, we use a helper array `used` with one entry per alternative, which is used to ensure that every ballot holds every alternative only once (Lines 5, 9 and 11). We assume a voting rule is given as a function f (Line 14) which gets a profile `prof` and the amount of voters `N` as parameters, and returns the set of winning alternatives as output. For the experiments within this section, the implementations of f will follow directly from their definitions.

After calling the test methods for the properties to be checked (in this case only Pareto dominance), we set a Boolean statement that is always false (Line 16) as content of C^{cons} (recognized by the solver by the keyword `assert`), which indicate to the solver that any program run that reaches this point is a counterexample of interest to us.

6.6 Definitions for the Experiments

We propose here to consider the axioms that characterize the Borda voting rule (Definition 3.2), as defined in a variant of Young's axiomatization (Cailloux and Endriss, 2016).

We also define a few axioms that are not satisfied by the Borda rule, and finally define two supplementary rules. All these concepts will be used in the experiments.

We repeat here the definition of the Borda rule from Section 3.2 for convenience:

Definition 6.19 (Borda rule) *The Borda rule, given a profile $(\gtrsim_i)_{i\in N}$, associates to each alternative a and voter i the score $s(a,i)$ equal to the amount of alternatives that a beats in $\gtrsim_i$, and associates to each alternative a the score $s'(a) = \sum_{i\in N} s(a,i)$. The winners are the alternatives that have the maximal score: $f_{Borda}((\gtrsim_i)_{i\in N}) = \arg\max_{a\in\mathcal{A}} s'(a)$.*

6.6.1 An Axiomatization of the Borda Rule

We now require a few further definitions.

A profile $(\gtrsim_i)_{i\in N}$ is *elementary* if and only if it has exactly two voters, and if the set of alternatives can be partitioned into disjoint subsets $A^{\text{top}}, A^{\text{bottom}} \subseteq \mathcal{A}$ with $A^{\text{top}} \cup A^{\text{bottom}} = \mathcal{A}$ and $A^{\text{top}} \neq \emptyset$ such that both voters have all alternatives in A^{top} preferred to all alternatives in A^{bottom}, and the voters have inverse preferences over A^{top}, and inverse preferences over A^{bottom}. Thus, denoting the voters by 1 and 2, and denoting by $\gtrsim|_A$ the restriction of $\gtrsim$ to A, $\forall A \in \{A^{\text{top}}, A^{\text{bottom}}\} : \gtrsim_1|_A = (\gtrsim_2|_A)^{-1}$. Let $T(R)$ denote the "top alternatives" of an elementary profile, meaning, the set of alternatives corresponding to A^{top} (this is legal as it is unique).

Example 6.20 (Elementary profile) *The profile R shown below, composed of the linear orders (a,b,c,d) and (b,a,d,c), is an elementary profile corresponding to $A^{top} = \{a,b\}$, with $\mathcal{A}$ equal to $\{a,b,c,d\}$.*

$$R = \begin{matrix} a & b \\ b & a \\ c & d \\ d & c \end{matrix}. \tag{6.5}$$

Given a linear order $\gtrsim \in \mathcal{L}(A)$, with $\mathcal{A}$ of size m, define the cycle corresponding to $\gtrsim$ as the set of m pairs consisting of the pairs of alternatives (a,b) such that b immediately succeeds to a in $\gtrsim$ (the rank of b is by one greater than the rank of a), union the pair of alternatives (z,a) where z and a are respectively the minimal and maximal elements of $\gtrsim$. For example, the cycle corresponding to the linear order (a,b,c) is $\{(a,b),(b,c),(c,a)\}$. Observe that, given any $\gtrsim \in \mathcal{L}(A)$, the cycle corresponding to $\gtrsim$ corresponds to exactly m linear orders in $\mathcal{L}(A)$. For example, the cycle $\{(a,b),(b,c),(c,a)\}$ also corresponds to (b,c,a) and (c,a,b).

We say that a profile $(\succsim_i)_{i\in N}$ is cyclic if and only if it has exactly m voters and m different linear orders, and some cycle corresponds to all its linear orders (equivalently, the linear orders in $(\succsim_i)_{i\in N}$ are all those that correspond to a given cycle).

Example 6.21 (Cyclic profile) *The profile R shown below is a cyclic profile corresponding to the cycle $\{(a,b),(b,c),(c,d),(d,a)\}$ with $\mathcal{A} = \{a,b,c,d\}$.*

$$R = \begin{array}{cccc} a & b & c & d \\ b & c & d & a \\ c & d & a & b \\ d & a & b & c \end{array}. \tag{6.6}$$

Below is the axiomatization that we use for the Borda rule, composed of the axioms Elem, Cycl, Canc and Reinf. It is very similar, but not identical, to the axiomatization given by Hobart Peyton Young (1974). The proof that these four axioms characterize the Borda rule is given in Cailloux and Endriss (2016).

Definition 6.22 (Elementary axiom) *The elementary axiom (Elem) mandates that the rule, when given any elementary profile, selects its top alternatives: $S_{Elem}(R) = \{T(R)\}$ if R is an elementary profile and $S_{Elem}(R) = \mathscr{P}^*(\mathcal{A})$ otherwise.*

Definition 6.23 (Cyclic axiom) *The cyclic axiom (Cycl) requires the rule to select all alternatives as tied winners when given any cyclic profile: $S_{Cycl}(R) = \{\mathcal{A}\}$ if R is a cyclic profile and $S_{Cycl}(R) = \mathscr{P}^*(\mathcal{A})$ otherwise.*

Definition 6.24 (Cancellation axiom) *The cancellation axiom (Canc) constrains the set of winners to be $\mathcal{A}$ when all pairs of alternatives (a,b) are such that a is preferred to b for as many voters as b is to a in $(\succsim_i)_{i\in N}$.*

Definition 6.25 (Reinforcement axiom) *As defined above in Reinf (Definition 6.18).*

6.6.2 Two Axioms Not Satisfied by Borda

Borda notoriously fails (when $|\mathcal{A}| \geq 3$) to satisfy the following two functional properties Cond and Maj, as well as the derived property Wmaj.

Given a profile $R = (\succsim_i)_{i\in N}$, we say that an alternative a obtains a strict majority against a', denoted by $a\ M_R\ a'$, if and only if more than half of the voters prefer a to a' in $(\succsim_i)_{i\in N}$: $a\ M_R\ a' \Leftrightarrow |\{\, i \mid a \succsim_i a' \,\}| > |\{\, i \mid a' \succsim_i a \,\}|$. An alternative is a Condorcet winner if and only if it obtains a strict majority against all other alternatives. Any Condorcet winner is unique.

Definition 6.26 (Condorcet property) *The Condorcet property (Cond) mandates that if there is a Condorcet winner, it becomes the sole winner.*

Definition 6.27 (Majority property) *The majority property (Maj) requires that, whenever some alternative is placed first by more than half of the voters in $(\succsim_i)_{i\in N}$, it becomes the sole winner.*

The Condorcet property is stronger than the majority property, i.e., $S_{\text{Cond}} \subseteq S_{\text{Maj}}$. Based on Maj, we define a weaker property, the weak majority property Wmaj, for which $S_{\text{Maj}} \subseteq S_{\text{Wmaj}}$.

Definition 6.28 (Weak majority property) *The weak majority property (Wmaj) requires that, whenever some alternative is placed first by more than half of the voters in $(\succsim_i)_{i\in N}$, it becomes a (not necessarily unique) winner.*

6.6.3 Two Condorcet Compatible Voting Rules

To end this section, we define two famous voting rules that satisfy the Condorcet property from Section 3.2.

Definition 6.29 (Black's rule) *The Black (1958, p. 66) rule selects the Condorcet winner if there is one, otherwise, the Borda winners.*

Given a profile $R = (\succsim_i)_{i\in N}$, let $M_R(a)$ denote the set of alternatives against which a obtains a strict majority, and ${M_R}^{-1}(a)$ the set of alternatives that obtain a strict majority against a.

Definition 6.30 (Copeland rule) *The Copeland (1951) rule (actually a close variant of a rule proposed by Ramon Llull in the 13th century (Colomer, 2013)), given a profile R, gives to each alternative the score $s(a) = |M_R(a)| - |{M_R}^{-1}(a)|$, and lets the alternatives with maximal score win.*

6.7 Experiments

In this section, we first experiment with the Borda rule itself. In Section 6.7.1, we look at whether the solver is able to find out that the Borda rule satisfies Pareto dominance, depending on the bounds we set on the amounts of alternatives and voters. We then check how long it takes to find an example that illustrates Pareto dominance (one where some

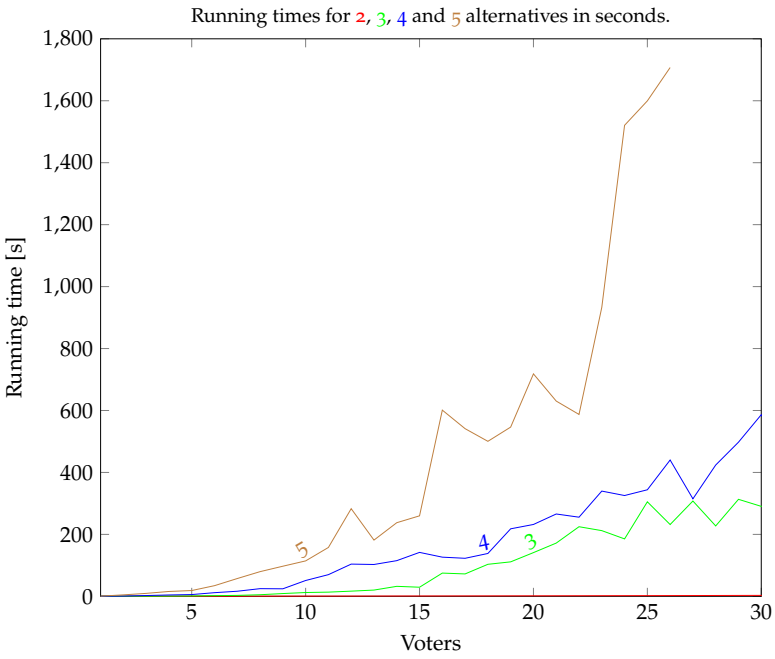


Figure 6.2: Running times for the automatic verification of Pareto dominance for the Borda voting rule.

alternative is dominated and indeed not included among the winners). In Section 6.7.2, we search for counterexamples that illustrate that the Borda rule fails to satisfy the properties defined in Section 6.6.2.

Second, we experiment with the Borda axiomatization. We suppose we are given a rule f, different from Borda (Definition 3.2), as a C function: in our experiments we consider the Black's rule and the Copeland rule. In this simple experimental setup, we thus know that f fails to satisfy at least one of the Borda axioms from Section 6.6.1. For each of these rules, we illustrate that we can find out automatically which axioms f fails to satisfy, and output a short proof of this, easy for a human to inspect. We also analyze in which situations we can prove that axioms are satisfied by f, for those that f satisfies.

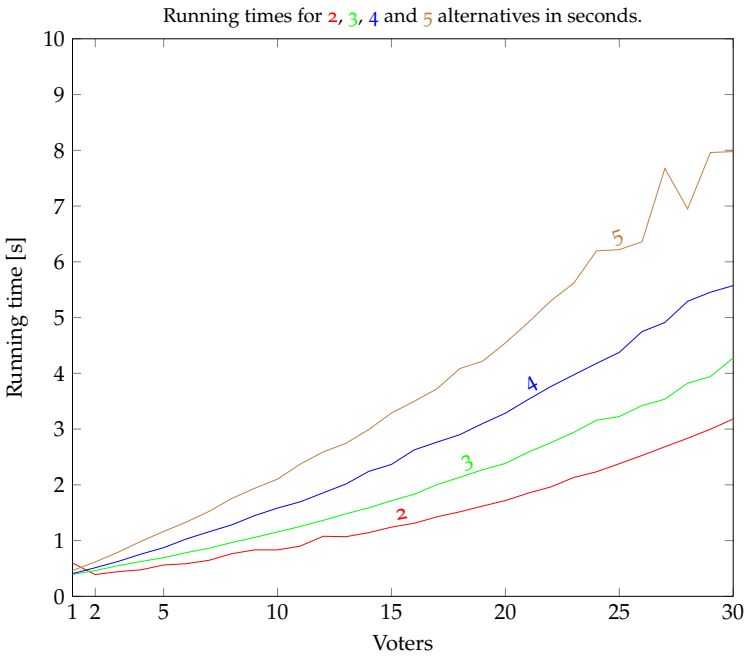


Figure 6.3: Running times for finding an illustration of Pareto dominance for the Borda rule.

6.7.1 Borda and Pareto Dominance

In Figures 6.2 and 6.3, we illustrate the sizes of situations, i.e., amounts of voters (on the x-axis) and alternatives (each having a different plot), that we are able to analyze. In this example, as well as in the following experiments, we only consider running times below 30 minutes to be reasonable, and stop computations which require more time than 30 minutes. Running times are given in seconds (on the y-axis) for up to 30 voters and 5 alternatives. Figure 6.2 shows the running times for the verification that Pareto dominance (Listing 6.2) holds for the Borda rule (the line for 2 alternatives is almost superposed to the x-axis). Experiments for 5 alternatives and more than 26 voters took more than 30 minutes and are thus not plotted.

Figure 6.3 shows the running times for finding an example situation that illustrates that Borda satisfies Dom. We used the program code from Listing 6.2 with the only difference that we negated the statement in Line 17. The function now indicates to the solver that it must find two different alternatives `bad` and `good`, such that every voter prefers `good` to `bad`, and the alternative `bad` is not a winner of the election.

In the first experiment, the solver proves that no profiles satisfy the provided conditions. We can see that the verification is feasible for small sizes of profiles within 15 minutes. Expectedly, in situations where some profile exists that satisfies the provided conditions, the running times are significantly smaller. They stay well below 10 seconds for this

example. Both experiments indicate that our method yields reasonable running times for at least up to five alternatives and 25 voters.

6.7.2 Counterexamples to Borda

We illustrate here our approach by finding counterexamples which show that the Borda rule does not satisfy the properties defined in Section 6.6.2 (those properties are satisfied by both Black's and the Copeland voting rule, however).

We start with the stronger Condorcet property Cond. The smallest example proving that Borda fails Cond can be found in less than one second for three voters and three alternatives:

$$R = \begin{matrix} c & c & b \\ b & b & a \\ a & a & c \end{matrix} \tag{6.7}$$

For the profile R, the Borda rule elects the alternatives $\{a, c\}$ instead of the Condorcet winner c.

Whereas an isomorphic counterexample is found in less than one second for the failure of Maj, we find two smallest examples (one regarding the amount of alternatives, and another one regarding the amount of voters) proving the failure of Wmaj.

For a minimal amount of alternatives, we find the smallest proof for three alternatives and five voters in less than one second:

$$R = \begin{matrix} a & a & a & b & b \\ b & b & b & c & c \\ c & c & c & a & a \end{matrix} \tag{6.8}$$

For the profile R, the Borda Borda!rulerule elects the alternative b instead of the majority winner a.

When searching a proof with a minimal amount of voters, we find the smallest proof for four alternatives and three voters in less than one second:

$$R = \begin{matrix} d & d & c \\ c & c & a \\ a & b & b \\ b & a & d \end{matrix} \tag{6.9}$$

For the profile R, the Borda rule elects the alternative c instead of the majority winner d. All given examples can be easily inspected manually.

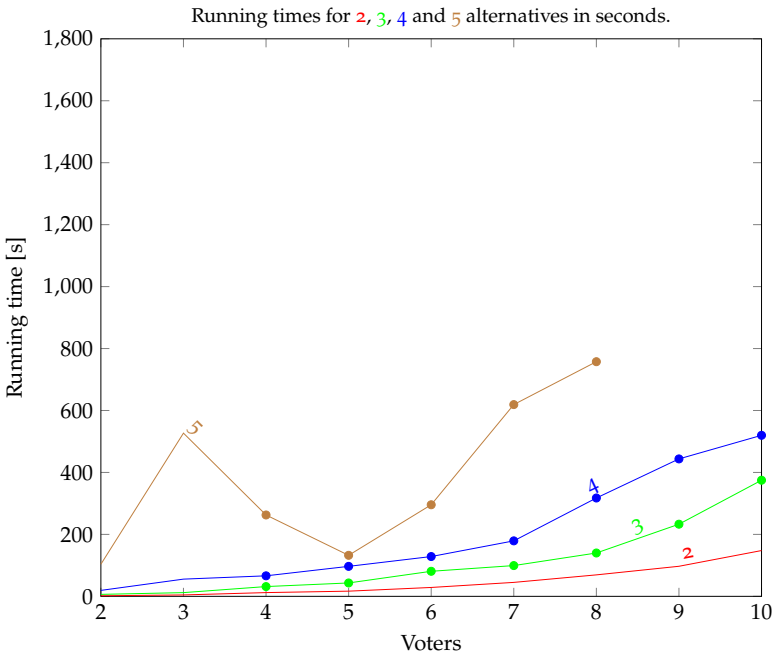


Figure 6.4: Running times for the verification of REINF for the Copeland voting rule.

6.7.3 Automatic Comparison of Borda with Other Voting Rules

We here want to compare Black's rule and the Copeland rule with the axiomatization of the Borda rule shown before. We first encode all four axioms in a way similar to the Pareto property shown in Listing 6.2. The axioms ELEM, CYCL and CANC are functional properties and can thus be encoded in the very same manner. As the axiom REINF is a 3-relational property, it needs further statements relating the profiles. The three profiles are initialized with symbolic nondeterministic values as seen in Listing 6.3. We consider the bound N as the amount of voters of the joined profile `prof`, and fix the sizes of the two sub-profiles `prof1` and `prof2` using a nondeterministic value `s` to define `prof1`'s size as `s`, and the size of `prof2` as `N - s`. We furthermore fix the profiles `prof1` and `prof2` with respect to `prof` using an array with nondeterministic values to model a mapping between the joined profile and its two sub-profiles.

Having encoded all four axioms as either functional or k-relational properties in C functions, we can now compare Black's rule and the Copeland rule with this axiomatization. We hereby rely on the *small-scope hypothesis* and focus on small profiles with up to five alternatives and eight voters.

Our method successfully verifies that both Black's rule and the Copeland rule satisfy the axiom ELEM in less than 17 seconds (for "small profiles," as defined above). As this axiom must only be verified for two voters, only the amount of alternatives must be bounded.

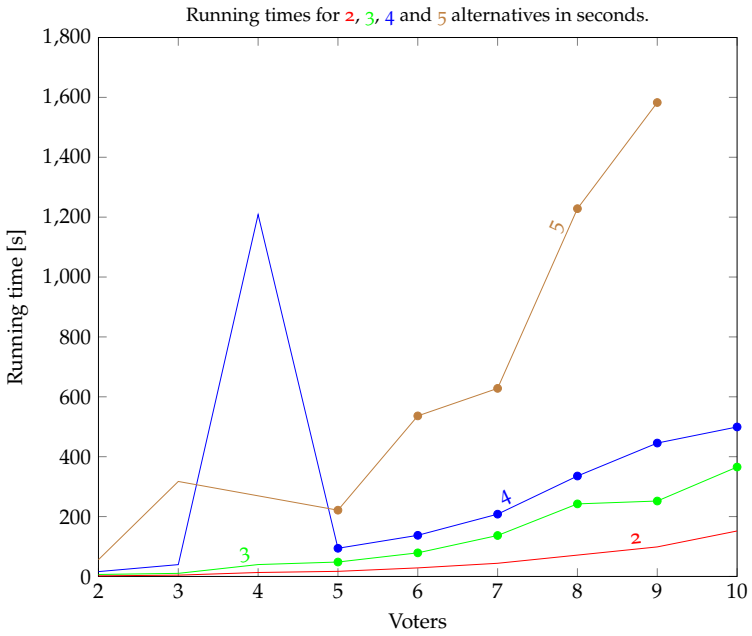


Figure 6.5: Running times for the verification of REINF for Black's voting rule.

For the axiom CYCL, the verification for both voting rules can be done within 10 minutes (also addressing only "small profiles" as defined above). Here, the amount of voters is fully determined by the amount of alternatives (CYCL only addresses profiles with an equal amount of alternatives and voters).

When checking the axiom CANC, for both rules, verification is achieved within a few minutes for the cases of eight voters or fewer when the amount of alternatives is bounded to three; five voters or fewer when the amount of alternatives is bounded to four; and three voters or fewer when the amount of alternatives is bounded to five. As running times for both rules are very similar, both are depicted in Table 6.2. Note that CANC trivially holds for odd numbers of n, and accordingly verification within the depicted range is achieved in less than 30 seconds.

For the axiom REINF, the running times can be seen in Figures 6.4 and 6.5 for the Copeland rule and Black's rule, respectively. Round dots indicate that violating voting situations have been found for the respective amounts of voters and alternatives. Missing parts (for the case of 5 alternatives) indicate a timeout as defined in Section 6.7.1. Nonetheless, the time for finding counterexamples of up to ten voters and five alternatives is significantly lower than for verifying that none exist (see, e.g., the running times for four and five alternatives in Figure 6.5, comparing running times for four and five voters). The smallest counterexamples can be found for both voting rules within less than one minute. These simultaneously serve as proofs, which are both short and easy for a human to inspect.

Table 6.2: Running times for the verification that Black's and Copeland rule each satisfy CANC.

m	n	verification time
≤ 3	≤ 8	$<$ 3 minutes
4	≤ 5 and 7	$<$ 2 minutes
4	6 and 8	$>$ 30 minutes
5	≤ 3 and 5 and 7	$<$ 1 minute
5	4 and 6 and 8	$>$ 30 minutes

The smallest example (in the amount of alternatives) proving that Black's voting rule fails REINF has been found in 48 seconds for three alternatives and five voters:

$$R_1 = \begin{matrix} a & c \\ c & a \\ b & b \end{matrix}, \quad R_2 = \begin{matrix} c & b & a \\ b & a & c \\ a & c & b \end{matrix}. \tag{6.10}$$

The elected alternatives for the profiles R_1 and R_2 are $\{a, c\}$ and $\{a, b, c\}$, respectively. For the joined profile $R_1 \cup R_2$, Black's voting rule elects the Condorcet winner a instead of the set $\{a, c\}$ mandated by REINF.

The smallest example, both in the amount of alternatives and in the amount of voters, proving that the Copeland rule fails REINF has been found in 32 seconds for three alternatives and four voters:

$$R_1 = \begin{matrix} b & a \\ a & c \\ c & b \end{matrix}, \quad R_2 = \begin{matrix} a & b \\ b & a \\ c & c \end{matrix}. \tag{6.11}$$

The elected alternatives for the profiles R_1 and R_2 are a and $\{a, b\}$ respectively. For the joined profile $R_1 \cup R_2$, the Copeland rule elects the set $\{a, b\}$ instead of only alternative a, which would be required by the REINF property.

Both examples can be easily inspected manually.

6.8 Efficient Verification via Program Transformations

In the previous sections, we have presented various examples for the verification of voting rules with respect to social choice properties using software bounded model checking. For a more convenient instrumentation of the SBMC tool, we have written the

prototypical tool BEAST[1] that translates the given parameters automatically so that the SBMC tool can process it, thereby avoiding repetitive user tasks.

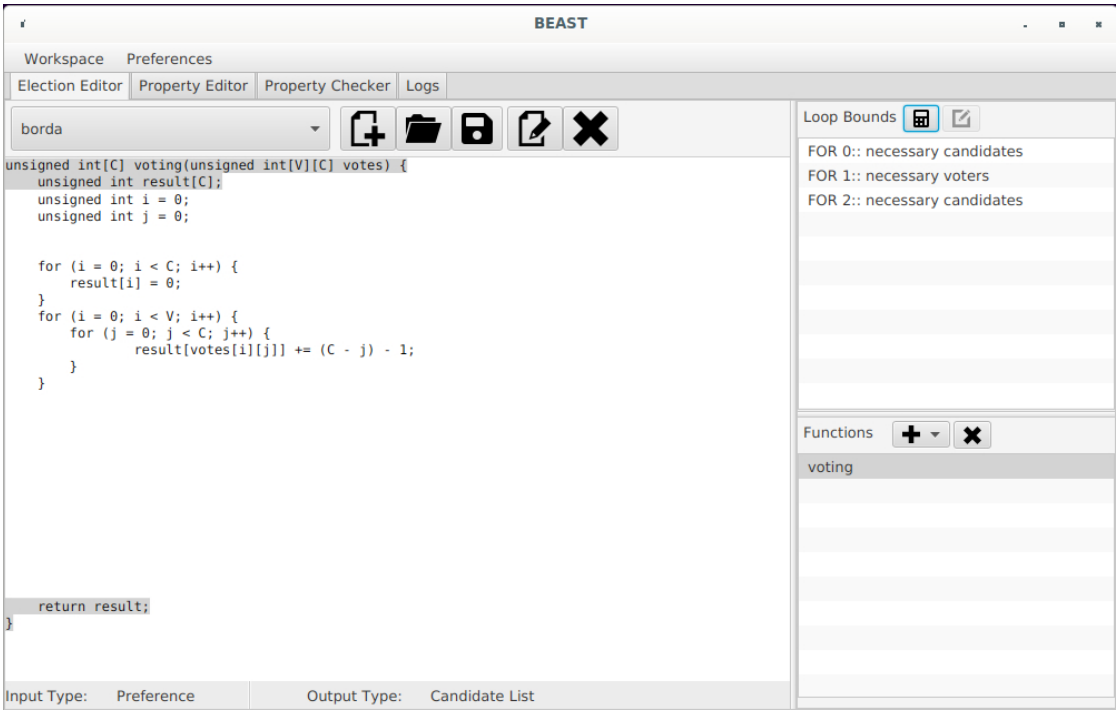


Figure 6.6: Implementation of Borda within BEAST.

In the following, we briefly report on the verification process for the example of the Borda rule and the reinforcement property REINF from Section 6.5. In Figure 6.6, we see the implementation of the Borda rule as a simplified C program as defined by Definition 3.2 in the *election editor* tab. Since we do not require the full generality possible in the C language, it suffices to provide 10 lines of code for the Borda rule, as well as the selection of appropriate input and output types (which are shown in the status bar below the implementation). Note that the constants `C` and `V` are constants for the amounts of alternatives and voters, respectively, that are afterward in the verification process filled in with the bounds that are chosen by the user. Within BEAST, we implemented a simple C parser that automatically detects loops, for each of which the respective bounds can be either directly inferred or specified by the user. On the right-hand side, we see that BEAST is able to automatically infer the loop bounds for this kind of loop. BEAST knows about the constants for alternatives and voters and hence presents the detected bounds as "necessary alternatives" and "necessary voters" for the according loops. Upon selection of the individual loop entries, we can also overwrite the computed values.

[1]The runnable source code, documentation of the property language, and the example below, are available under https://github.com/VeriVote/BEAST.

Note that the method signature as well as the return statement at the end are shaded in gray, as these are fixed based on the previous selection of input and output types. In this case, these are preference profile and list of alternatives (which contains all – possibly tied – winners), but other types can be selected as well. In case we require auxiliary functions, these can be added by the menu below the computation of the loop bounds on the right-hand side. When starting the verification, BEAST generates the rest of the necessary methods, including a main method, that are necessary for the verification process.

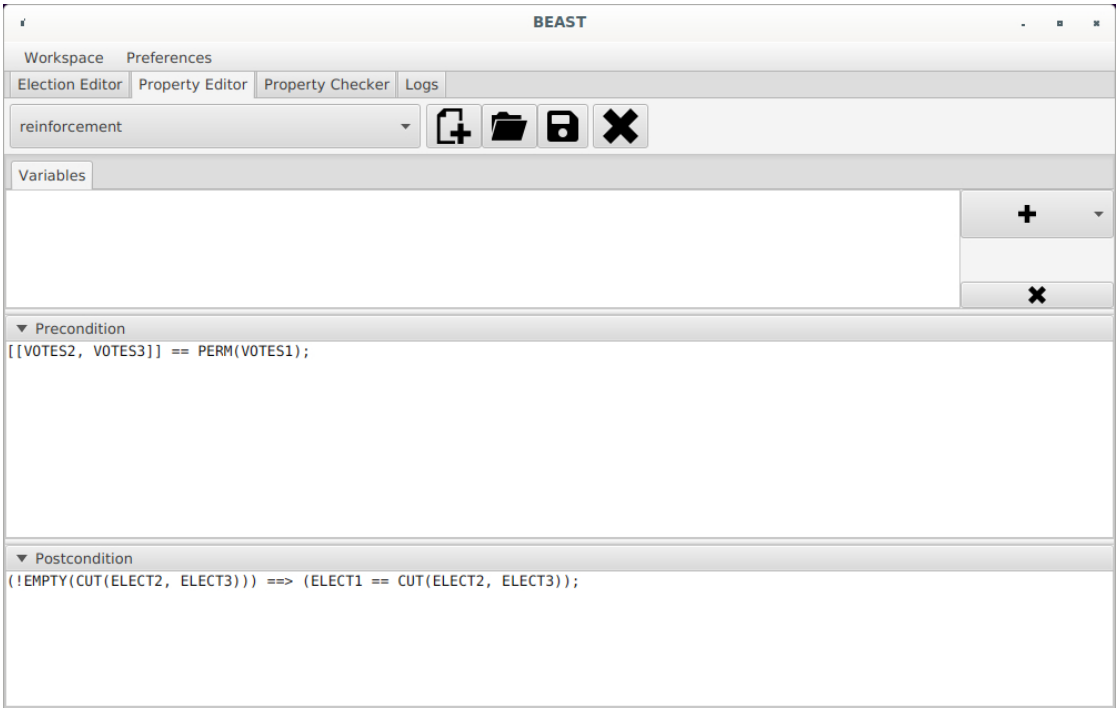


Figure 6.7: Implementation of reinforcement property within BEAST.

Besides the implementation, we need to specify the property which we want to verify. This can be done in the *property editor*, which we see in Figure 6.7 for the reinforcement property as defined in Definition 6.18. For a convenient way to specify properties, we defined a simple language with patterns that denote common parts of social choice properties. Moreover, the upper *variables'* menu allows defining the required nondeterministic values when we want to address, e.g., a specific voter or alternative; in this case none are necessary. Below the *variables'* menu, there is both the *precondition* menu and the *postcondition* menu. Therein, we have access to the keywords `VOTES` and `ELECT` which are templates for the input and the output of the voting rule, in our case a preference profile and a list of alternatives, respectively. For expressing properties between multiple executions of the voting rule, i.e., relational properties, we complement the keywords with a natural number that refers to the execution to which they belong. For example, `VOTES1` and `ELECT1` suffice when

specifying functional properties, and we can use VOTES2 and ELECT2 to address input and output of the second execution, and so on. In the background, BEAST generates the required method calls, instructions, and C structures accordingly.

For the reinforcement property, we use the precondition to specify for the three different profiles VOTES1, VOTES2, and VOTES3, that the concatenation of VOTES2 and VOTES3 is a permutation of the profile VOTES1. Note that PERM is a pattern provided by BEAST that, in the background, translates to a nondeterministic array and an assumption that specifies the permutation relation to the array constructed on the left. By means of using numbers up to 3, BEAST already knows that it must generate three calls to the voting rule.

Finally, the postcondition specifies an implication, that whenever the intersection (specified by CUT) of the election results ELECT2 and ELECT3 is not empty (! denotes negation and EMPTY a predicate that evaluates to *true* for an empty profile), then the election result ELECT1 of the first execution contains exactly this intersection. BEAST performs the whole translation in the background and generates a "raw" C program including assertions and assumptions that can then be analyzed by CBMC.

We do not elaborate on the remaining parts of BEAST for performing the verification and presenting counterexamples at this point, as the essential parts thereof have already been covered in Section 6.5.

6.9 Summary

In this chapter, we have presented and formally defined an automatic approach to argue for and against voting rules based on axiomatic social choice properties. The approach is fully automated and based on the software analysis technique *bounded model checking*. Our case study on the Borda, Copeland, and Black's voting rule shows that illustrative proofs are obtained automatically in reasonable time for small amounts of voters and alternatives. This might appear surprising given the combinatorial structure of preference profiles and set-valued outcomes.

Based on the small-scope hypothesis, we argue that our approach can be used for arguing about voting rules, especially because it provides short and human-readable proofs when it detects that a voting rule fails to satisfy some axiom.

Ideas for future work include extensions to reasoning about rules that have only incomplete specification. For example, we could automatically conceive an argument (in the form of an example profile and a set of winners, as illustrated in the experiments) that would attack any rule that does not satisfy a given axiom. In this way, it would be possible to also argue against classes of rules, in addition to concrete rules as illustrated in this chapter. Our approach could also be extended to automatically illustrate differences

between two sets of axiomatic properties without the need for any explicit voting rule. A more ambitious extension would consist in automatically deriving proofs that a winner is a "right" winner by using several example profiles, as Cailloux and Endriss (2016) have illustrated by means of a non-automatic procedure.

A longer-term objective is to mix the ideas proposed in this work and further optimization techniques (Beckert, Bormer, Kirsten, et al., 2016) with elicitation procedures in order to obtain a system that would permit to recommend a voting rule, possibly based on work in elicitation of scoring rules (Cailloux and Endriss, 2014) and on work that defines justified recommendations as those that resist counterarguments (Cailloux and Meinard, 2020).

7

Computation of Dependable Election Margins for Reliable Audits

> Beware of bugs in the above code; I have only proved it correct, not tried it.
>
> Donald Ervin Knuth, *Notes on the van Emde Boas construction of priority deques: An instructive use of recursion*, 1977

Commonly for high-stakes elections in which some stakeholder suspects some irregularity or any kind of problem within the processing of the ballots and the computation of the tally, the regulations often allow them to demand a reliable auditing procedure that scrutinizes the election result.

One reliable method to create confidence in the outcome of an election among the electorate is to audit the election result against the physical evidence, i.e., the ballots. Different methods for auditing elections exist, some of them require the computation of a margin, that is the minimal amount of ballots to be changed, misfiled, etc. to affect the election outcome. For those methods, the precise definition of the margin is often hidden inside the theory, as it depends on the election function – or social choice function – and the particular auditing methodology. This means, that (1) for many election functions, including ranked-choice voting (RCV) and Single-Transferable Vote (STV), or election functions that combine different electoral systems, for example on state and federal level, it is difficult if not impossible to give closed forms for how to compute a margin, and (2) even if one manages to find a closed form for how to compute the margin, the implementations of election function and margin computation differ, for example in

the way ambiguities are resolved, when and how to which precision to round, how tiebreaking rules are implemented, etc.

In this chapter, we focus on auditing methods that require the margins to be known before they can be applied. Examples of these methods are, e.g., risk-limiting audits that draw a random sample of paper ballots (Section 3.3) whose size is computed from (a) a risk limit, i.e., how confident we wish to be in the election result, and (b) the margin. For a *comparison audit*, the margin of a risk-limiting audit is defined as the minimal amount of votes that would need to be misfiled in order to change the election outcome. The margin is identical to the amount of votes that would have had to be miscounted or tampered with during tabulation. If the election margin is large, only a small sample needs to be drawn and audited. The smaller the margin, the larger the sample. In the worst case, the audit will trigger a full manual recount.

We describe a way to compute the margins that does not presuppose the existence of a closed form for the margin and works directly on the source code (e.g., written in C/C++). Our technique can be applied to any election function, but it will perform best on those that are conceptually simple, such as D'Hondt and Saint-Laguë. The technique can in principle also be applied to more complex election functions, such as instant-runoff voting (IRV), but only for small elections with a small amount of seats and alternatives.

In the following, we introduce our method that, based on SBMC (Section 2.2), allows to automatically compute election margins for arbitrary election functions, and apply it to the well-known D'Hondt method for elections on German state level in Section 7.1. An extension that leads to increased efficiency is described in Section 7.3. In Section 7.4, we present a case study where we apply our method to compute the election margin for the main part of the 2015 Danish national parliamentary elections.

The content of this chapter has been previously published by Beckert, Kirsten, Klebanov, et al. (2017) and Beckert, Bormer, Goré, et al. (2017).

7.1 Efficient Computation of Election Margins

We assume that an election function is given as an imperative program (a C function called `election_function` in our case) as well as a concrete input (denoted as `vote_table`) for that election function. The `vote_table` is the result of vote counting and tabulation. We model `vote_table` as an integer array of size `PARTIES`, where `PARTIES` is the amount of different stacks into which identical votes are accumulated during counting.

The idea of our approach is to use an SBMC tool to check an assertion claiming that, when `vote_table` is changed by putting at most a certain amount m of votes on other stacks than they were on, the outcome of the election is *not* changed. If that assertion is

```
1  void verify() {
2      int new_votes[PARTIES], diff[PARTIES], total_diff, pos_diff;
3      for (int i = 0; i < PARTIES; i++) {
4          diff[i] = nondet_int();
5          __CPROVER_assume (-1 * MARGIN ≤ diff[i] ≤ MARGIN);
6          __CPROVER_assume (0 ≤ ORIG_VOTES[i] + diff[i]);
7      }
8
9      for (int i = 0, total_diff = 0, pos_diff = 0; i < PARTIES; i++) {
10         new_votes[i] = ORIG_VOTES[i] + diff[i];
11         if (0 < diff[i]) pos_diff += diff[i];
12         total_diff += diff[i];
13     }
14     __CPROVER_assume (pos_diff ≤ MARGIN);
15     __CPROVER_assume (total_diff == 0);
16
17     int *result = election_function(new_votes);
18     assert (equals(result, ORIG_RESULT));
19 }
```

Listing 7.1: Implementation of the margin computation for CBMC.

provable, we know that the actual election margin is greater than m. If the assertion is not provable, we know that the actual election margin is less than or equal to m. In the latter case, the SBMC tool generates a counterexample to the assertion demonstrating that the electionoutcome can be changed by changing m votes. Having this proof obligation as a basis, we can use binary search to find a value for m such that the assertion holds for $m - 1$ but fails for m, i.e., m is exactly the election margin.

The check for a particular prospective margin m can be executed by running the SBMC tool CBMC on the program shown in Listing 7.1, where the variables written in capital letters are given as concrete input values, and the method `nondet_int()` is a CBMC feature in order to denote nondeterministic, i.e., potentially different for each function call, and symbolic, i.e., unknown, integer values. The changes in the sizes of the vote stacks are nondeterministically chosen (Line 4) in such a way that the total difference is zero (assumption in Line 15), i.e., votes can be moved from one stack to the other but not removed or created, and such that the amount of votes in each stack cannot become negative (Line 6). Other types of margins for other kinds of changes to the vote table can be computed using different assumptions on the chosen values for `diff`. The changes are added to the original vote table for computing the new table (Line 10). Then, the election result for the new vote table is computed by calling the method `election_function` (Line 17).

Algorithm 7.1 Binary search for election margin using SBMC.

Input:

`election_function`: implementation of the election function
`ORIG_VOTES`: array with the size of each of the stacks of identical votes, i.e., the input for the election function
`PARTIES`: size of the vote table array

Output:

`MARGIN`: computed election margin

```
1  function searchMargin
2      ORIG_RESULT ← election_function(ORIG_VOTES)          // initialization
3      MARGIN ← 0
4      left ← 0
5      right ← ∑_{i=1,...,PARTIES} ORIG_VOTES[i]            // total amount of votes
6      while left < right do                                // search for margin
7          MARGIN ← left + ⌈(right−left)/2⌉
8          result ← cbmc(verify(), MARGIN, PARTIES, ORIG_VOTES, ORIG_RESULT)
9          if result = SUCCESS then
10             left ← MARGIN + 1, MARGIN ← MARGIN + 1
11         else
12             right ← MARGIN
13         end if
14     end while
15     return MARGIN
16 end function
```

Finally, the program contains the assertion to be checked by CBMC (Line 18), expressing that the new election result is equal to the original one. Intuitively, we have encoded any difference between the original election outcome and the new one as a bug to be found by the model checker. This also means that our approach gives us a concrete redistribution of votes for the computed margin, as CBMC encodes detected bugs as concrete paths through the program, which lead to the assertion violation, i.e., the changed outcome.

The algorithm performing a binary search for the exact election margin is shown in Algorithm 7.1 (for our experiments we use a shell script implementation of this algorithm). The algorithm takes as input the implementation of an election function and a concrete vote table. Its output is the exact election margin. The algorithm first calls `election_function` to obtain the original election result (Algorithm 7.1). The left and right bounds of the binary search are initialized to zero, respectively the total amount of votes (Algorithms 7.1

to 7.1). Then, a while-loop (Algorithms 7.1 to 7.1) performs the binary search and calls CBMC on the program from Listing 7.1 with different values for `MARGIN`, i.e., different alternatives' margins, until the solution is found. If the result of CBMC indicates that `MARGIN` is too low, the left bound is increased (Algorithm 7.1), and if CBMC indicates that `MARGIN` is either the correct margin or is too high, then the right bound is decreased (Algorithm 7.1). To be more precise, if the result of calling CBMC reads `SUCCESS`, we know that the assertion in the program in Listing 7.1 holds, i.e., the election outcome cannot be affected, and the speculative margin `MARGIN` is too low; otherwise `MARGIN` either is the correct election margin or it is too high.

Note that neither the algorithm in Algorithm 7.1 nor the program in Listing 7.1 make any further assumptions regarding the election function. Our method can be applied to arbitrary implementations of `election_function` without making any changes, only influencing the computation time needed by the satisfiability solver used as a backend, e.g., for more complex mathematical operations. The approach can also be adapted to more complex ballot structures. And, as said above, margins for different notions of vote changes can be computed by using different assumptions on the array `diff` in Listing 7.1, and margins for different notions of changes in the election outcome can be computed by using different versions of the function `equal` called in Line 18 from Listing 7.1.

7.2 Margin Computation for the D'Hondt Method

Margin computation also plays a central role for risk-limiting audits regarding the results after performing seat apportionment methods such as the D'Hondt or Saint-Laguë method (Stark and Teague, 2014). In this section, we exemplarily apply our technique to the D'Hondt method, which allocates mandates to a number of parties based on the votes cast for these parties. Before the D'Hondt election function is applied, vote counting and tabulation sorts the votes into stacks, where each stack contains votes for a single party. The input for the election function then is the amount of votes for each party (i.e., the amount of votes in the corresponding stack).

The D'Hondt method proportionally allocates mandates to parties in such a way that the amount of votes represented by mandates is maximized, i.e., the votes-per-seats ratio – intuitively the price in the amount of votes to be paid by a party to get one seat – is made as high as possible while still allocating all seats in parliament. By this means, D'Hondt achieves an – as far as possible – proportional representation in parliament (Gallagher, 1991).

D'Hondt can be implemented as a *highest-averages* method: the amount of votes for each party is divided successively by a series of divisors, which produces a table of quotients (or averages). In that table, there is a row for each divisor and a column for each party.

```
1  int *election_function(int vote_table[PARTIES]) {
2      int *mandates = malloc(PARTIES * sizeof(int));
3      int divisor[PARTIES];
4
5      for (int i = 0; i < PARTIES; i++) mandates[i] = 0;
6      for (int i = 0; i < PARTIES; i++) divisor[i] = 1;
7
8      int elected = 0;
9      for (int j = 0, j < MANDATES; j++) {
10         for (int i = 0; i < PARTIES; i++)
11             if (divisor[i] * vote_table[elected]
12                    < divisor[elected] * vote_table[i]) elected = i;
13         mandates[elected]++;
14         divisor[elected]++;
15     }
16     return mandates;
17 }
```

Listing 7.2: Implementation of the D'Hondt method as a C program.

For the D'Hondt method, these divisors are the natural numbers $1, 2, \ldots,$ `MANDATES`, where `MANDATES` is the total amount of mandates to be distributed. Then, the greatest numbers in the quotient table – respectively the parties in whose columns these numbers are – are each allocated one seat. The "final" seat goes to the `MANDATES`'th greatest number. Hence, the threshold level of the votes-per-seats ratio lies in the interval between the `MANDATES`'th greatest number and the (`MANDATES` + 1)'st greatest number of all computed averages in the quotient table.

An efficient C implementation of D'Hondt is shown in Listing 7.2. There, the constants `PARTIES` and `MANDATES` encode the amounts of parties and the amount of mandates to be allocated, respectively. The input is given in the array `vote_table`, which holds the amounts of votes cast for each individual party. This implementation avoids constructing the complete quotient table. Instead, the program stops as soon as the `MANDATES`'th greatest quotient has been found. For this purpose, the divisors currently under consideration for finding the next highest value are stored in the array `divisor` for each party. Note that in case of a tie, the order in `vote_table` is the tiebreaker, i.e., the first party in `vote_table` which is tied with the current maximum divisor takes the seat.

After initializing the arrays `mandates` and `divisor` (Lines 5 and 6), we execute the outer loop (Lines 9 to 15) `MANDATES` times. Each time, the program iterates the inner loop (Lines 10 to 12) to find the maximum

$$\texttt{elected} \quad = \quad \max_{i=1,\ldots\texttt{PARTIES}} \frac{\texttt{vote\_table}[i]}{\texttt{divisor}[i]}$$

Table 7.1: Preliminary official results for the 2005 Schleswig-Holstein elections.

Party	Votes	%	Mandates	%
Christian Democratic Union (CDU)	576 100	42.1	30	43.4
Social Democratic Party (SPD)	554 844	40.6	29	42.0
Free Democratic Party (FDP)	94 920	6.9	4	5.8
Alliance '90/The Greens	89 330	6,5	4	5.8
South Schleswig Voter Federation (SSW)	51 901	3.7	2	2.9
Totals	1 367 095		69	

and then assigns one seat to the `elected`'th party (Line 13), and increases the divisor for that party (Line 14). To find the maximum, the comparison

$$\texttt{vote\_table[elected]}/\texttt{divisor[elected]} < \texttt{vote\_table[i]}/\texttt{divisor[i]}$$

is replaced by

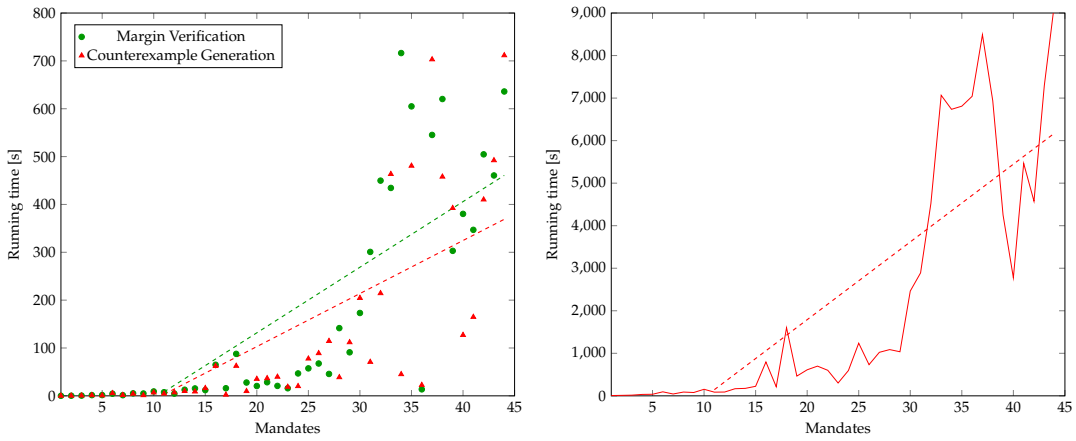
$$\texttt{divisor[i]} * \texttt{vote\_table[elected]} < \texttt{divisor[elected]} * \texttt{vote\_table[i]},$$

which is equivalent as the divisors are positive numbers. The advantage of using the latter form for the comparison is to avoid dealing with fractional numbers and rounding effects in C. This is a sensible choice for any implementation of D'Hondt as, depending on the programming language and hardware, rounding may both show unexpected behavior and potentially lead to faulty election results.

In order to test our margin computation for D'Hondt, we use the preliminary official results of the Schleswig-Holstein state elections in 2005.[1] In that election, 1, 367, 095 votes were cast and 69 mandates were to be allocated. Out of the 13 parties running, four parties received the necessary quota of 5% to be eligible for the mandate allocation. The fifth party to receive seats, the South Schleswig Voter Federation, represents the Danish minority and is exempted from the quota rule for reasons of minority protection. The mandates (seats in parliament) were allocated using the D'Hondt method. The parties, their votes, and the allocated mandates are shown in Table 7.1.

We apply our approach to the vote numbers (i.e., the `vote_table`) of the Schleswig-Holstein election for various values of `MANDATES`. In doing so, we are able to compute the margin of the election with the running time increasing for higher values of `MANDATES` as shown in Section 7.2 and Section 7.2. The running time for the final check is shown in Section 7.2. This check requires showing that the election result can be changed by changing m votes (counterexample generation) but cannot be changed by changing $m - 1$ votes (margin

[1]The results of that election are also used as an example in the German Wikipedia article on the D'Hondt method (http://de.wikipedia.org/wiki/D'Hondt-Verfahren).



(a) Time for last step in computation.

(b) Accumulated time for whole computation.

Figure 7.1: Running times of automatic margin computation for the D'Hondt method with various values for `MANDATES`.

verification), implying that m is the true margin. Section 7.2 shows the accumulated time for the complete binary search that computes m. For values of `MANDATES` between 2 and 45, the computed margins range from only 433 (for `MANDATES` = 23) to 177,863 (for `MANDATES` = 2). Note that, with only two mandates, the CDU and the SPD each get a seat; the margin of 177,863 then is the amount of votes that have to be moved from the SPD to the CDU so that the CDU gets both mandates instead of only one, which is smaller than the amount of votes that would have to be moved from the SPD to the FDP so that the FDP gets a seat instead of the SPD.

The running times shown in the figure do not form a smooth curve because they depend on the margin that is computed, which is, e.g., smaller for 40 mandates than for 35. Yet, the numbers increase with the value of `MANDATES`. Moreover, as can be seen from the figure, they get prohibitively large for more than about 45 mandates.

Thus, our approach can be applied to real implementations of real election functions, but only if the number of loop iterations does not go beyond a few hundred (about 5 parties times 45 mandates in this case). For elections with a larger amount of parties and mandates or election functions with more complex loop nestings, improvements are required. One such improvement is discussed in the following section.

7.3 Automated Finding of Election Parameters

The election function defined by the D'Hondt method can also, equivalently, be described without a quotient table. Instead, a quota is chosen, i.e., an amount of votes needed to

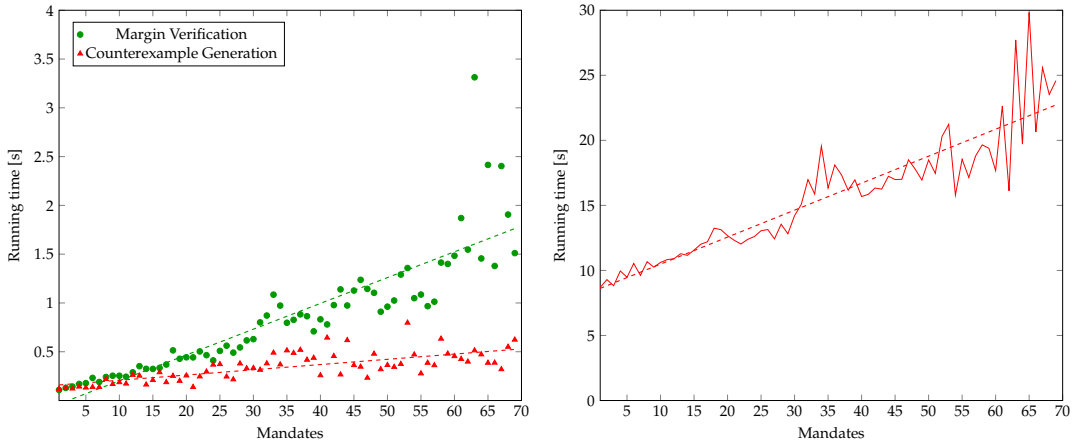
```
1  int *election_function(int votes[PARTIES]) {
2      int *mandates = malloc(PARTIES*sizeof(int));
3      for (int i = 0; i < PARTIES; i++) mandates[i] = 0;
4
5      int quotaNumerator   = nondet_int();
6      int quotaDenominator = nondet_int();
7
8      __CPROVER_assume (0 < quotaNumerator   ≤ INT_MAX);
9      __CPROVER_assume (0 < quotaDenominator ≤ MANDATES);
10     __CPROVER_assume (quotaDenominator < quotaNumerator);
11
12     for (int i = 0; i < PARTIES; i++) {
13         __CPROVER_assume (0 ≤ quotaDenominator * votes[i] ≤ INT_MAX);
14         mandates[i] = ((quotaDenominator * votes[i]) / quotaNumerator);
15         __CPROVER_assume (0 ≤ mandates[i] ≤ MANDATES);
16     }
17
18     int total_mand = 0;
19     for (int i = 0, total_mand = 0; i < PARTIES; i++)
20         total_mand += mandates[i];
21     __CPROVER_assume (total_mand == MANDATES);
22
23     return mandates;
24 }
```

Listing 7.3: Implementation of the Jefferson method as a symbolic C program.

"buy" one mandate, such that the resulting mandates per party, when rounded down to the next natural number, sum up to the required total amount of mandates. This is known as Jefferson's method and is similar to *largest-remainder* methods such as the Hare-Niemeyer method. The quota corresponds to the lowest quotient in the D'Hondt table for which a mandate is allocated.

If the implementation of an election function is based on choosing or searching for some parameter (here the quota), then the margin computation can be made much more efficient by replacing the search for the parameter by a nondeterministic choice to be resolved by the SBMC tool.

An implementation of the Jefferson method in C is shown in Listing 7.3. The program uses a nondeterministic choice of $\texttt{quota} = \texttt{quotaNumerator}/\texttt{quotaDenominator}$ (Lines 5 to 6). Assumptions are made to limit the range of the quota (Lines 8 to 10 and Line 13). The amount of mandates for each party is computed (Line 14), as well as the total amount of mandates (Lines 18 to 20). Then, the assumption is checked that the total amount of mandates for the chosen quota is the correct one (Line 21). This final check is an



(a) Time for last step in computation.

(b) Accumulated time for whole computation.

Figure 7.2: Running times of automatic margin computation for the Jefferson method with various values for `MANDATES`.

assumption and not an assertion, i.e., we want to consider only the case(s) where the total amount of mandates is correct; other cases are irrelevant. An assertion, on the other hand, would have to be true for all cases where the (other) assumptions are fulfilled. Note that this implementation does not deal with tiebreaking, as in this case no such quota can be found, and no program execution path can satisfy the assumption in Line 21. However, tiebreaking mechanisms can easily be integrated in the program.

The running times of the automatic margin computation for the 2005 Schleswig-Holstein state elections with various values for `MANDATES`, i.e., the total amount of mandates to be allocated, are shown in Section 7.3 and Section 7.3. Note that these running times are much lower than those for the D'Hondt method in Section 7.2 and Section 7.2. Now, all computations stay well below the timeout of 9,000 seconds (i.e., 2.5 hours), even below 30 seconds. Moreover, the computation of the election margin for the original amount of mandates in the election, which is 69, is now easily possible; that margin is 634. The computed margins range from only 42 (for `MANDATES` = 62) to 177,863 (for `MANDATES` = 2). Performing our method for various values for `MANDATES` scales well on the Jefferson method, as we got rid of the loop depending on the value of `MANDATES`. However, further experiments also indicate a non-exponential dependency on the value for `PARTIES`. For example, an allocation of 69 mandates to 10 parties takes about 55 seconds, whereas for 20 parties, the analysis runs in ca. 300 seconds.

Naturally, the implementation in Listing 7.3 cannot be compiled and executed to produce a binary file using standard C compilers, because it contains constructs only understood by the model checker CBMC. However, it can nevertheless be compiled and executed using CBMC, which also allows for performing tests and similar measures in order to

Table 7.2: Official results for the 2015 national Danish elections (Danmarks Statistik, 2015).

Party	Votes	%	Mandates	%
Socialdemokratiet	924 940	26.3	43	31.9
Radikale Venstre	161 009	4.6	2	1.5
Det Konservative Folkeparti	118 003	3.4	0	0.0
SF – Socialistisk Folkeparti	147 578	4.2	2	1.5
Liberal Alliance	265 129	7.5	9	6.7
Kristendemokraterne	29 077	0.8	0	0.0
Dansk Folkeparti	741 746	21.1	33	24.4
Venstre, Danmarks Liberale Parti	685 188	19.5	33	24.4
Enhedslisten – De Rød-Grønne	274 463	7.8	10	7.4
Alternativet	168 788	4.8	3	2.2
Totals[1]	3 515 921		135	

generate confidence in the implementation. Furthermore, when any C implementation of the Jefferson method is given, it is easy to construct a CBMC version uniformly by replacing the search for `quota` by a nondeterministic choice. The same principle for making margin computations more efficient can uniformly be applied to any election function where parameters such as quotas are chosen or computed within the election function.

7.4 Evaluation for the National Danish Elections

In this section, we demonstrate the applicability of our approach to a further, more complex real-world election, namely the Danish parliamentary elections in 2015. The Danish elections use a two-tier system, further classified as an *adjustment-seat system*, where the main part of the seats (135 mandates) is allocated using the D'Hondt method for each of the lower-tier electoral districts (so-called constituencies) separately (Elklit, Pade, and Nyholm Miller, 2011). The remaining seats (40 mandates) are used for adjusting the proportionality with respect to the three higher-tier districts using the Saint-Laguë method (which is also a *highest-averages method*, bounded by the Hare quota).

The aggregated results for the 2015 election are shown in Table 7.2. For the sake of readability, the table only contains the total amounts of votes, not the amounts for each constituency. In the following, we perform our analysis on the first tier, i.e., the distribution of the 135 mandates which are allocated separately within each constituency.

[1]Excluding non-party votes.

Table 7.3: Results for Danish constituency Sjællands Storkreds (Danmarks Statistik, 2015).

Party	Votes	%	Mandates	%
Socialdemokratiet	146 464	27.9	7	35.0
Radikale Venstre	16 906	3.2	0	0.0
Det Konservative Folkeparti	15 083	2.9	0	0.0
SF - Socialistisk Folkeparti	20 575	3.9	1	5.0
Liberal Alliance	32 598	6.2	1	5.0
Kristendemokraterne	1 996	0.4	0	0.0
Dansk Folkeparti	134 195	25.6	6	30.0
Venstre, Danmarks Liberale Parti	102 818	19.6	4	20.0
Enhedslisten - De Rød-Grønne	35 374	6.7	1	5.0
Alternativet	18 202	3.5	0	0.0
Totals[1]	524 211		20	

Using the Jefferson-version of D'Hondt, we compute a margin of 10 votes within 7,815 seconds, i.e., around 2 hours and 10 minutes. The final verification (proving that a change in 9 votes cannot change the election outcome) takes 53 seconds and a counterexample for 10 votes (i.e., an example ballot box that does change the election outcome) can be found within 27 seconds. The generated counterexample shows that shifting – only – 10 votes from *SF – Socialistisk Folkeparti* to *Venstre, Danmarks Liberale Parti* in the constituency of Sjællands Storkreds results in a different election outcome where one mandate goes the same way as the 10 votes. That is, SF loses its single seat, and Venstre then has five seats. The vote table and election results for the constituency of Sjællands Storkreds are shown in Table 7.3.

With the table-based D'Hondt method as a basis (Listing 7.2), the margin computation takes 16,860 seconds (around 4 hours and 40 minutes). The final verification takes 659 seconds and a counterexample can be found within 652 seconds. Using the table-based D'Hondt implementation, for which margin computation is less efficient, is possible in this case because the amount of mandates for each constituency is sufficiently low (around 20).

[1]Excluding non-party votes.

7.5 Summary

In this chapter, we have presented a method that computes election margins fully automatically. It can be applied to arbitrary implementations of election functions without understanding or even knowing how the election result is computed. Our approach can be applied to real implementations of real election functions if the amount of loop iterations in the election function does not go beyond a few hundred. With the improvement from Section 7.3 for guessing parameters needed in the computation, the method scales up to larger and more complex elections.

Future work includes the computation of different types of election margins and an integration with software for supporting real-world risk-limiting audits. Further, we plan to apply our method to election functions for which margin computation is notoriously hard (such as instant-runoff voting). First experiments indicate that such functions are hard for our method as well. However, it will be possible to adapt our method to computing lower bounds for margins in IRV elections using techniques described in the literature (Cary, 2011; Sarwate, Checkoway, and Shacham, 2013).

Part IV

Secure Election Management Systems

Technically, the city of Ankh-Morpork is a Tyranny, which is not always the same thing as a monarchy, and in fact even the post of Tyrant has been somewhat redefined by the incumbent, Lord Vetinari, as the only form of democracy that works. Everyone is entitled to vote, unless disqualified by reason of age or not being Lord Vetinari.

Terry Pratchett, *Unseen Academicals*, 2009

8

Security Verification of the GI Voter-Anonymization Software

> RIGGED 2020 ELECTION: MILLIONS OF MAIL-IN BALLOTS WILL BE PRINTED BY FOREIGN COUNTRIES, AND OTHERS. IT WILL BE THE SCANDAL OF OUR TIMES!
>
> Donald John Trump, *Twitter Communication*, 2020

While the previous parts of this thesis are in principle applicable to both analog and electronic voting systems, real-world voting systems are oftentimes at least to a certain extent realized as electronic systems. As described in Section 3.4, there are some desirable requirements such as end-to-end verifiability that cannot be realistically realized in purely analog voting systems reliably. This chapter, for this matter, deals with the election management part of the *Polyas 3.0 E-Voting System* that has been used in the annual elections by the GI in the years 2019 and 2020. The Polyas 3.0 E-Voting System provides universally verifiable tallying, a form of participation privacy, as well as protection against ballot stuffing. We specifically analyze the part of the system that anonymizes voter identities by the generation of secure voter credentials, which is to be carried out by an independent entity such as the election council.

Within this chapter, we report on how we formally verify that the voter credentials are generated and processed securely and attacks such as the unauthorized injection of additional ballots to the ballot box, i.e., *ballot stuffing*, is banned, given the software is executed as specified. We describe the provided guarantees in Section 8.1, the e-voting

system in Section 8.2, the voter credential generation tool in Section 8.2, the verified property, its specification and formal verification in Section 8.3.

The content of this chapter has not been previously published, while the idea is described in a short paper by Beckert, Brelle, et al. (2019) and the results have been presented at the 2019 annual meeting by the GI working group on formal methods and software engineering for secure systems (FoMSESS).

8.1 Electronic Voting and Secure Voter Credentials

From the general requirements that are concerned with end-to-end verifiability described in Section 3.4, we start by describing the specific properties that the Polyas 3.0 E-Voting System is specified to provide. These are namely, universally verifiable tallying, ballot and participation privacy, as well as protection against ballot stuffing.

Universally Verifiable Tallying. This property denotes that the voting system provides means to guarantee that the complete tallying process, i.e., the process starting from the content of the ballot box and the content of the registration board until the publication of the election results, has been done correctly, and the election result correctly reflects the given content of the ballot box.

Universal verifiability is guaranteed by zero-knowledge proofs that are produced during the tallying phase. In the tallying phase, the collected votes are first mixed in a secure shuffle on the basis of Wikström's verifiable mix net (Haines, 2019), and then, once we cannot learn anything from the order of the votes anymore, securely decrypted by the election provider Polyas with the public voter credentials, before the decrypted ballots are finally tallied. The shuffle and the decryption step both produce zero-knowledge proofs which prove that the initial votes correspond to the shuffled ones, and that the shuffled votes correspond to the decrypted ones. The position paper by Beckert, Brelle, et al. (2019) describes that for the secure shuffle and decryption, independent parties developed verification tools to check the zero-knowledge proofs provided during those steps. The verification tools have been employed by the independent parties under close supervision of the GI election council during the ceremony when the results were computed.

Ballot Privacy. Ballot privacy denotes the property that the content of the ballots is kept secret, i.e., that it cannot be observed who any given voter actually voted for. For the current configuration, Polyas may learn the identifiers of voters who actually voted, but given that these identifiers cannot be linked to actual voter identities, ballot privacy is preserved.

The ballots are already encrypted on the voter-side based on the voters' secure credentials. Originally, the secure voter credentials are generated by the central registrar – in our setting the GI election council – who passes the credential passwords in an encrypted form together with the voters' addresses, but not their public identifiers, to the printing facility for their dissemination. Subsequently, the printing facility sends these secure credentials to the voters via mail, guaranteeing ballot privacy throughout this process. Since only encrypted ballots are cast, Polyas is also not able to link individual voters to their cast votes during the tallying phase. The registrar also sends the public voter credentials to Polyas, and makes sure not to put them together with the passwords, which is a crucial privacy requirement.

Protection Against Ballot Stuffing. This property denotes that, even though Polyas is in control of the ballot box in order to publish the correctly authorized encrypted ballots, Polyas cannot misuse this control for injecting additional illegitimate ballots without the risk of getting detected during by the procedure for universal verification. However, this protection cannot prevent Polyas from (hypothetically) throwing away some votes.

As described above, the registrar makes sure not to give the private voter credentials, i.e., the passwords, to the registry (Polyas). From this assumption, Polyas is not able to generate any additional votes, since votes that were not generated by any of the secret credentials are thrown out later during the tallying phase, since the zero-knowledge proofs only consider votes that were generated legitimately.

8.2 Elections of the German Society for Computer Scientists

The Polyas 3.0 E-Voting System (Truderung, 2019; Truderung, 2021), as used in the GI 2019 elections, consists on the top-level of three phases, registration phase, voting phase, and tallying phase. In the following, we briefly describe the election process, before zooming in on the registration phase for our case study in the remaining sections of this chapter. The process is visualized in Figure 8.1.

Registration Phase. The registration phase involves the registrar (the GI election council), the registry (Polyas), and the printing facility. In this phase, the election council first uses the software for credential generation in order to translate a list of all eligible voters and their addresses into a generated list of the voters' private credentials (their passwords) and the corresponding public credentials (roughly, the verification keys which are signed by the private credentials). The public credentials are then uploaded to the voting system and published on the registry board, and the private credentials are

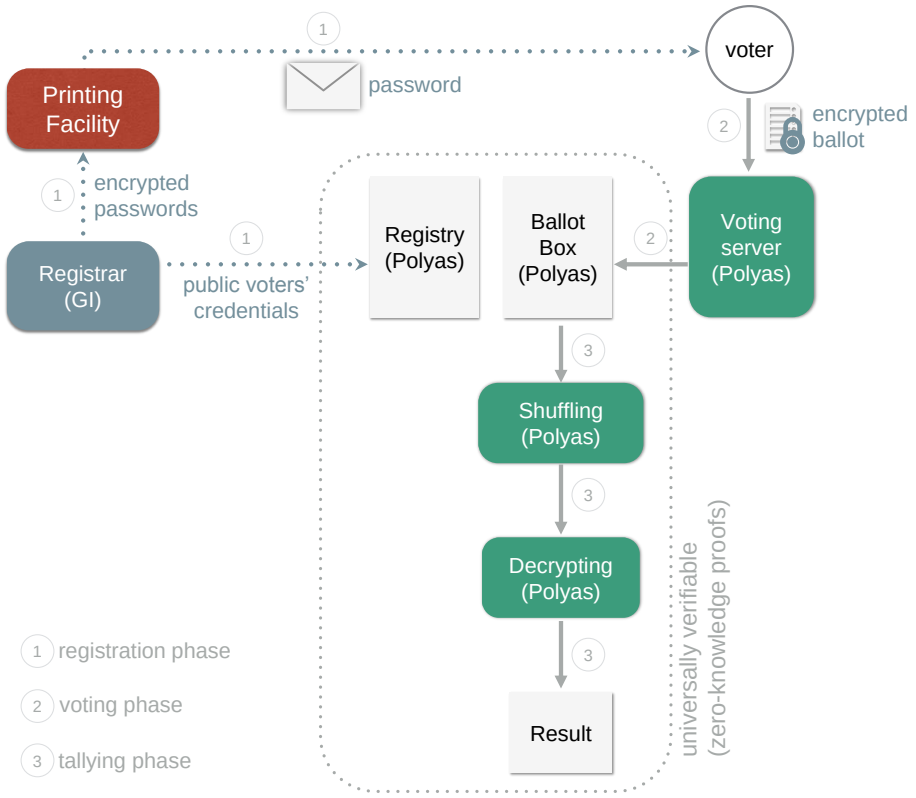


Figure 8.1: All phases in the Polyas 3.0 E-Voting system.

encrypted and sent to the printing facility, which decrypts them and sends them to the voters' addresses.

Voting Phase. In the voting phase, the voters, the voting server, and the ballot box are involved. The voting server and the ballot box are provided by Polyas. Here, the voters use their credentials to authenticate themselves on the voting platform via the election web page, and fill out their ballot in their browser. Once the voter has finished filling out their ballot, the voting client creates an encrypted and signed ballot from the voter's inputs and sends the ballot to the voting server, who adds it to the ballot box. It is important that the encryption and signing happens on the client side, such that no unencrypted private information is transferred to the server.

Tallying Phase. Finally, the tallying phase involves only the registrar (Polyas) and the ballot box. Here, the encrypted ballots are mixed and decrypted in a secure and verifiable way by producing zero-knowledge proofs for the correct shuffle and a correct decryption

(see Section 8.1). Finally, the decrypted votes are tallied and the computed election results then published. As described above, the registration phase provides crucial security guarantees that are required for ensuring the properties described in Section 8.1 in the other two phases. The detailed process is visualized in Figure 8.2.

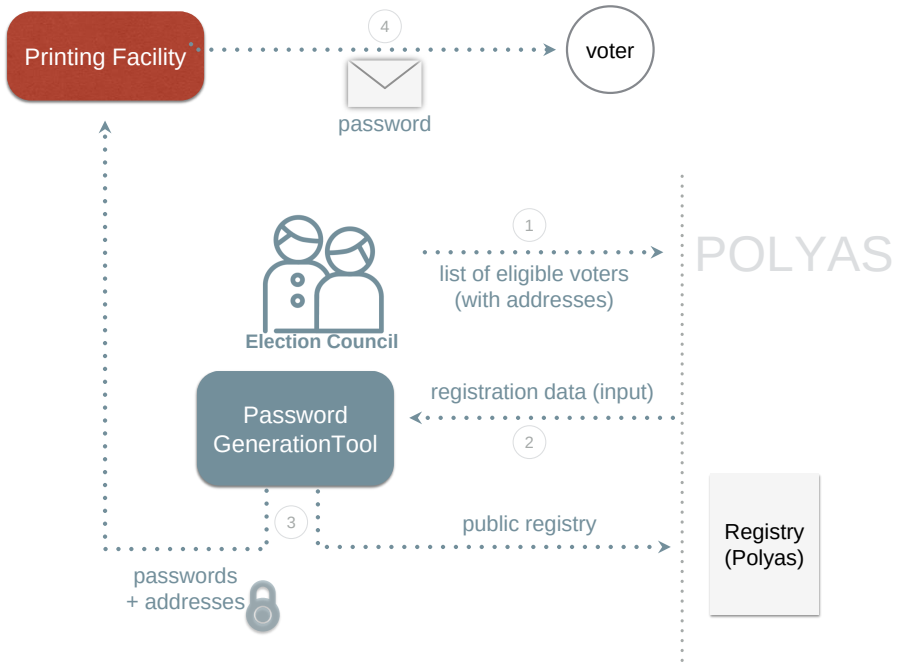


Figure 8.2: Voter registration phase in the Polyas 3.0 E-Voting system.

For this matter, the e-voting system employs a tool that generates the secure credentials and separates the data that will be sent to the voter from the data sent to Polyas. This is actually the property which we want to formally verify, i.e, that the data sent to Polyas is processed independently of the secret credentials for the voters. The original software is written in Kotlin, but for the verification using the KeY system, we translated the Kotlin code to Java code. The full Java program consists of roughly seven classes with 30 program methods in 780 lines of code.[1] The simplified functionality which we are interested to verify is shown in Listing 8.1.

[1]The source code together with the specifications and proofs is available under https://github.com/VeriVote/polyas-core3-open-cred.

```
1  public class CredentialTool {
2    ...
3    File generate(RegistrationData input) {
4      List polyas = new List();
5      List printingFacility = new List();
6      for (VoterRecord it : input) {
7        String password = Crypto.randomCredential80();
8        VoterData generatedData =
9          CredentialGenerator.generateDataForVoter(it.voterId,
10                                                  password);
11       printingFacility.add(generatedData.password);
12       polyas.add(generatedData.voterID,
13                  generatedData.hashedPassword,
14                  generatedData.publicSigningKey);
15     }
16     return new File(printingFacility, polyas);
17   }
18   ...
19 }
```

Listing 8.1: Simplified Password Generation Software

Therein, the method `generate` receives the registration data as input and outputs a file that consists of two separate data sets, one for the printing facility and one for Polyas. The method starts by initializing the two respective lists for the data sets, and then iterates over each voter record. For each record, a new random password is generated and further processed to generate public and private credentials from the voter data. Then, the password is added to the data set for the printing facility, and the voter ID, the hashed password and the public signing key are added to the data set for Polyas. Finally, the two data sets are returned as method output.

8.3 Verification in the KeY System

Having described our target functionality, we use the information-flow contracts described in Section 2.3 to fully specify the noninterference property of interest, requiring about 420 lines of specification.[1] This corresponds to a ratio of lines of specification per lines of code of roughly 0.54. Using information-flow contracts, we provide contracts for all involved methods as well as the methods called from libraries, such as the method `randomCredential80()` for the generation of a random password seen in Listing 8.1.

[1]The fully specified source code and the proofs are available for inspection under https://github.com/VeriVote/polyas-core3-open-cred/-/tree/key-hash-oracle.

```
1  public final class Hashes {
2    ...
3    private static int currentIndex;
4    private static BigInteger[] VALUES;
5
6    /**  Computes the uniform hash of the provided data. */
7    /*@ requires 0 <= currentIndex && currentIndex < VALUES.length;
8      @ ...
9      @ assignable currentIndex;
10     @ determines \result.value
11     @   \by currentIndex,
12     @   (\seq_def int i; 0; VALUES.length; VALUES[i].value);
13     @*/
14   public static BigInteger uniformHash(BigInteger upperBound,
15                                        String s1, String s2,
16                                        String s3) {
17     return VALUES[currentIndex++];
18   }
19 }
```

Listing 8.2: Ideal Hash Functionality in Java

Note that, as can also been seen in the method `generate`, Polyas receives a hashed password. Since the hashed password obviously depends on the password, we can only establish real noninterference if we replace the method that hashes the password by a so-called *ideal* functionality. The reasoning that this ideal functionality is safe to assume must happen outside KeY, similarly as has been done for the noninterference verification of the *sElect* system by Küsters, Truderung, Beckert, et al. (2015).

The implemented ideal functionality for computing the hashed value is shown in Listing 8.2. Therein, the method `uniformHash` simply ignores the input parameters and instead fills up a static array with incremental values. The method contract specifies that, provided the incremented index is within the bounds of the array, the method guarantees that the result value only depends on the index and the array itself. In a nutshell, this means that the hash function is only one-way.

8.4 Summary

Provided this simpler ideal functionality, the full proof required about 980000 rule applications which were mostly applied automatically, except for 0.06 percent of rule applications that had to be done interactively. The entire specification and verification process took around two to three person weeks. Besides the ideal functionality, the proof

holds under the assumptions that the employed Java classes cannot be overridden, the attacker cannot observe the heap size or amount of created objects during program execution, and the provided input is well-formed and actually contains the public key from Polyas.

Future work includes a seamless reasoning on the ideal functionality, to increase the trust in our justification that it is indeed a trustworthy implementation for our setting.

Part V

Related Work and Conclusion

I rarely end up where I was intending to go, but often I end up somewhere I needed to be.

Terry Pratchett, *The Long Dark Tea-Time of the Soul*, 1988

9

Related Work

> It is time to unmask the computing community as a Secret Society for the Creation and Preservation of Artificial Complexity.
>
> Edsger Wybe Dijkstra, *The next fifty years*, 1996

Having described formal methods targeted at trustworthy voting systems for three principal components between the election layer and the computational layer, this chapter now reviews some work by others that relates to the methodological approaches used in this thesis.

We start by reviewing formal methods within the field of secure multi-party computation aimed at the verification of protocols and the construction of secure circuits in Section 9.1. Then, Section 9.2 looks at other works that target social choice properties for voting rules by modular verification or some kind of reliable construction or synthesis. More broadly, Section 9.3 reviews work that has been done on the static analysis of voting rules targeting relational properties for verification, or exploiting symmetry structures for finding counterexamples. Further work on the task of computing election margins is reviewed in Section 9.4. Finally, Section 9.5 gives a brief review on related work regarding the verification of secure information flow on software level.

9.1 Protocol Verification and Construction of Secure Circuits

Our work on the generation of secure card-based cryptographic protocols is the first work which applies formal methods to the field of card-based cryptography. However, a large range of research has been done using formal methods in the more general field of secure two-party and multi-party computations. This can be clustered into either analyzing security protocols given as high-level, abstract (and usually idealized) models, or program-based approaches targeting real(istic) protocol (software) implementations. Avalle, Pironti, and Sisto further structure this into the two main approaches of automated model extraction and automated code generation. We refer the interested reader to overviews as given by Blanchet (2012) or Avalle, Pironti, and Sisto (2014), and only go into a few selected works for which we identified closer links to our approach, e.g., using software bounded model checking (SBMC), SAT solvers on real(istic) protocol implementations, or relating to the analyzed security model. Standard cryptographic assumptions using lower-level computational models are – albeit more realistic – usually harder to formalize and automate. One notable line of research is CBMC-GC (Franz et al., 2014) which builds on top of the tool CBMC (Clarke, Kroening, and Lerda, 2004). CBMC-GC uses SBMC in a compiler framework translating secure computations of ANSI C programs into an optimized Boolean circuit which can subsequently be implemented securely utilizing the garbled circuit approach. Another similar setting to ours is analyzed by Rastogi, Swamy, and Hicks (2019), who also assume an *honest-but-curious* attacker model. Therein, a domain-specific language is built on top of the F* language, a full-featured, verification-oriented, effectful programming language by Swamy et al. (2016). Swamy et al. then implement MPC programs with enabled formal verification provided by the semantics of the language.

9.2 Modular Verification and Program Synthesis

We base the core component type in our verified construction framework on the electoral modules from the unified description of electoral systems by Grilli di Cortona et al. (1999). Therein, Grilli di Cortona et al. devise a complex component structure for describing hierarchical electoral systems with a focus on proportional voting rules including notions of electoral districts and concepts of proportionality. Note, however, that the component type within this work is already quite different from the structures by Grilli di Cortona et al. (1999). In the current state, essentially, both concepts only share the concept of reducing and partitioning the set of alternatives.

General informal advice on voting rule design is given by Taagepera (2002). Moreover, a first approach for composing voting rules in a limited setting is given by Narodytska, Walsh, and Xia (2012) that is readily expressible by our structures. Other work designs

voting rules less modularly for statistically guaranteeing social choice properties by machine learning (Xia, 2013). Prior modular approaches also target verification (Ghale et al., 2018; Verity and Pattinson, 2017) or declarative combinations of voting rules (Charwat and Pfandler, 2015), but ignore the social choice or fairness properties targeted by our work.

We have defined our compositional approach within Isabelle/HOL (Nipkow, Paulson, and Wenzel, 2002), a theorem prover for higher-order logic. Isabelle/HOL provides interactive theorem proving for rigorous systems design. Further work on computer-aided verification of social choice properties for voting rules using the theorem prover HOL4 has been done by Dawson, Goré, and Meumann (2015). More lightweight approaches with some loss of generality, but the merit of generating counterexamples for failing properties has been devised by Beckert, Bormer, Kirsten, et al. (2016) and Kirsten and Cailloux (2018). Therein, techniques for relational verification of more involved social choice properties have been applied. Another interesting approach has been followed by Pattinson and Schürmann (2015), where voting rules are directly encoded into HOL rules within tactical theorem provers.

9.3 Relational Verification, Symmetries, and Counterexamples

While we use program verification technology based on first-order logic for the automated verification of voting rules, there are other approaches using tactical theorem provers and higher-order logic. Examples are proofs carried out by Dawson, Goré, and Meumann (2015), Goré and Meumann (2014), and Pattinson and Schürmann (2015). Verification using tactical theorem provers may lead to even higher confidence levels, but the task is inherently difficult and time-consuming, resulting in huge and laborious interactive proofs.

Moreover, we translated first-order logic proof obligations for verification and counterexample generation using software bounded model checking for proving that is targeted for social choice properties. There are further approaches to translate proof obligations with first-order expressions in a more general setting, i.e., verification of method contracts for general Java programs by Beckert, Kirsten, Klamroth, et al. (2020), where the achieved bounds were significantly smaller than those obtained in our experiments.

There exists extensive research analyzing theoretical voting rules on a more intuitive or experimental level, involving empirical experiments or comparisons of previous elections (Gallagher, 2013; Brams and Fishburn, 1988; Regenwetter and Grofman, 1998). In addition, there is research on the verification of concrete voting systems, i.e., considering a concrete voting software (Dennis, Yessenov, and Jackson, 2008).

Furthermore, a multitude of theoretical work on proving and finding new incompatibilities of voting rule properties has been done using SAT solvers (Tang and Lin, 2009; Geist and Endriss, 2011; Brandt, Geist, and Peters, 2017; Chatterjee and Sen, 2014; Brandt and Geist, 2016). Another application of computer-aided techniques to social choice theory is the use of machine learning for designing new social choice mechanisms satisfying desired axiomatic properties (Xia, 2013).

We use weaved programs to reduce the required effort for relational verification to that of "standard" program verification. A general notion of product programs that supports such a reduction is provided by Barthe, Crespo, and Kunz (2011).

There is also related work on breaking symmetries regarding the problem specification (Mancini and Cadoli, 2005; Cadoli and Mancini, 2007) and on methods for automatically generating symmetry-breaking predicates for classes of combinatorial objects for search problems (Shlyakhter, 2007). More applied work on this task provides tools which detect symmetries in structured graphs generated from CNF formulae (Darga et al., 2004).

9.4 Margin Computation

Our work on the computation of dependable election margins is a *generic* method that infers election margins for any election function, for which an implementation is available. In contrast to our work which is a *generic* method as long as an implementation of the tallying procedure is available, there has been a lot of research on how to compute margins for *specific* tallying procedures, for which that problem is particularly hard. The most prominent example is instant-runoff voting (IRV) where margin computation is NP-hard (Bartholdi and Orlin, 1991). Methods for computing lower bounds on margins for IRV have been developed by Cary (2011) and Sarwate, Checkoway, and Shacham (2013); and methods for computing the exact margin have been presented by Magrino et al. (2011) and, more recently, by Blom et al. (2016).

The computation of the margin for an election is an instance of the general problem of inverting a function for which an implementation is given, i.e., to ask for an input to the implementation that leads to a particular kind of output. The idea of using model checkers for solving such problems has also been applied in the field of test-case generation, where one is interested in input values leading to some specific program behavior (Vorobyov and Krishnan, 2012). For example, the software bounded model checker CBMC has been integrated into the extensive test-suite FShell (Holzer et al., 2008). Similar techniques have been used for generating high-quality game content, such as well-designed puzzles that are hard to solve (A. Smith, Butler, and Popovic, 2013).

In the context of elections, SBMC with SAT or SMT solvers can furthermore be used for analyzing, whether the given tallying procedure does indeed compute the correct result with respect to some given formal criteria (Beckert, Goré, Schürmann, et al., 2014).

9.5 Implementation-Level Information-Flow Verification

We did our noninterference verification case study with the KeY verification system. Verification of information-flow control has typically been done using type systems, which are sound, but generally capture only properties that are more lightweight compared to noninterference (Banerjee and Naumann, 2005; Barthe, Pichardie, and Rezk, 2013). Further tools for analyzing information-flow control translate the program to program dependency graphs and apply efficient slicing techniques (Hammer and Snelting, 2009; Jürgen Graf, Hecker, and Mohr, 2013). They are sound, but produce many false alarms as they do not provide full precision.

Moreover, there are other information-flow verification approaches that are based on KeY in combination with more efficient techniques (Beckert, Bischof, et al., 2018; Küsters, Truderung, Beckert, et al., 2015; Beckert, Herda, et al., 2020). For our case study, however, their whole-program approach would not work due to the many library calls for the cryptographic functionalities within the Polyas system. Further verification approaches for cryptographic functionalities of Java-like programs haven been done by Fournet, Kohlweiss, and Strub (2011) and Küsters, Truderung, and Juergen Graf (2012). Finally, the symbolic execution engine within KeY has also been used for the automatic detection of information-flow leaks by Do, Bubel, and Hähnle (2017).

10 Conclusion

There is no real ending. It's just the place where you stop the story.

Franklin Patrick Herbert Jr., *Interview at California State College, Fullerton*, 1969

Within this dissertation, we have devised and evaluated targeted formal methods for voting systems, with contributions within each principal voting system component. The devised methods enable a systematic development of trustworthy voting systems which can be provenly verified. The application target of a development using our methods is a reliable voting software with formal guarantees that the given requirements are met, in a way that is comprehensible to informed users and domain experts.

The three considered principal components are the voter-ballot box communication, the election method, and the election management system. The formal methods are based on a variety of formal techniques with different degrees of expressiveness and automation. The devised formal methods provide a reliable bridge over traditional bounds between the election layer – containing understandable requirements – and the computational layer – which contains the runnable procedures – such that trust can be reliably obtained without the need to inspect the layers separately. The contributions of this thesis hence advance the development of trustworthy voting systems using reliable and explainable formal requirements such that (1) a domain expert may afterward reliably comprehend that the requirements are met, (2) selections of the given requirements may be reliably justified or compared to other requirements, and (3) the trust obtained in objectives (1) and (2) may be reliably scrutinized by other domain experts.

10.1 Summary

For this matter, we developed targeted formal methods from the design of abstract components that can be trusted towards executable and reliable software that can be audited dependably.

The distinct contributions of this dissertation are:

(I) a method for the generation of secure card-based communication schemes,

(II) a method for the synthesis of reliable tallying procedures,

(III) a method for the efficient verification of reliable tallying procedures,

(IV) a method for the computation of dependable election margins for reliable audits,

(V) a case study about the security verification of the GI voter-anonymization software.

The first component, the voter-ballot box communication channel, is addressed by contribution (I), for which we built a bridge from the communication channel to the cryptography scheme by automatically generating secure card-based schemes from a small formal model with a parameterization of the desired security requirements. We devised a new method to search card-based protocols for any secure computation, by giving a general formal translation applicable to be used by the formal technique of software bounded model checking (SBMC). This method allows finding new protocols automatically, and prove lower bounds on required shuffle and turn operations for any protocol, and provide an example for the computation of a minimal AND protocol. We extended our verification method to the case of decks using only two colors, which is more common in the field of card-based cryptography.

The second component, the election method, is addressed by contributions (II) to (IV). Within contribution (II), we built a bridge from the election method to the tallying procedure by a formal composition framework, which we instrumented to automatically synthesize a runnable tallying procedure from the desired requirements given as social choice properties. We evaluated our method on sequential majority comparison, which is used in many practical knock-out tournament settings. We devised a new method to systematically and automatically synthesize voting rules from compact composable modules to satisfy formal social choice properties. We devised composition rules for a selection of common social choice properties, such as monotonicity or Condorcet consistency, as well as for reusable auxiliary properties. By design, these composition rules give formal guarantees, in the form of an Isabelle proof based on the properties satisfied by the component properties, that a synthesized voting rule fulfills the social choice property of interest as long as its components satisfy specific properties, which we have proved within Isabelle/HOL for the scope of our case study. We extended

this approach by a synthesis tool which automatically synthesizes, given the desired properties, a suitable voting rule as a verified and directly runnable Scala program together with the checkable Isabelle proof. Our approach is applicable to the construction of a wide range of voting rules which use sequential or parallel modular structures, notably voting rules with tiebreakers, elimination procedures, or tournament structures. This includes well-known rules such as instant-runoff voting, Nanson's method, or sequential majority comparison (SMC). The approach can be flexibly extended with additional modules, compositional structures, and rules, for integration into voting rule design or verification frameworks.

For contribution (III), we devised a formal method that assists in automatically analyzing, evaluating, and comparing tallying procedures readily to allow re-adjusting the procedures and requirements, which we evaluated on various tallying procedures from the literature. We have presented and formally defined an automated approach to argue for and against voting rules based on axiomatic social choice properties. The approach is fully automatic and based on the software analysis technique *bounded model checking*. Our case study on the Borda, Copeland, and Black's voting rule shows that illustrative proofs are obtained automatically in reasonable time for small amounts of voters and alternatives. This might appear surprising given the combinatorial structure of preference profiles and set-valued outcomes.

By contribution (IV), we built a general formal method that computes election margins for a user-provided tallying procedure, which we evaluated for the Danish national parliament elections 2015. We have presented a method that computes election margins fully automatically. The method can be applied to arbitrary implementations of the tallying procedures without understanding or even knowing how the election result is computed. Our approach can be applied to real implementations of real tallying procedures if the amount of loop iterations in the tallying procedure does not go beyond a few hundred. With improvement for guessing parameters that are needed in the computation, the method scales up to larger and more complex elections.

Finally, the third and final component, the election management system, is addressed by contribution (V), for which we performed a case study on an anonymization strategy within a real-world e-voting system for the generation of secure voter credentials. The e-voting system has been employed in this form for the annual elections of the German Informatics Society (GI – "Gesellschaft für Informatik") in 2019 and 2020. We obtained a precise proof requiring only 0.06 percent of interactive rule applications. The proof holds under the assumptions that the employed Java classes cannot be overridden, the attacker cannot observe the heap size or the amount of created objects during program execution, and the provided input is well-formed and actually contains the public key from Polyas.

10.2 Outlook

Regarding the method for secure card-based protocols, it would be interesting and worthwhile to exploit symmetries within our model to further scale, e.g., to finite-runtime protocols for five or more cards. Moreover, it may be worthwhile to explore the mixing of different card settings (e.g., only distinguishable cards with one pair of identical cards), in order to find more efficient protocols.

For the construction and synthesis of voting rules that meet desired fairness properties, the current framework could be extended by modules and compositions for more complicated voting rules, in order to achieve a more flexible notion of composition, e.g., to support run-off voting rules where, e.g., surplus votes are transferred. Another sensible extension is the composition of voting rules based on *distance rationalization* that is very flexible, since almost all voting rules can be constructed this way with the caveat that the composition structures would be less imperative than the structures described herein.

Regarding efficient verification techniques and counterexample generation for voting rules, it might be interesting to reason about voting rules that are incompletely specified. For example, we could automatically conceive an argument (in the form of an example profile and a set of winners, as illustrated in the experiments) that would attack any rule which does not satisfy a given axiom. In this way, it would be possible to also argue against classes of rules, additionally to concrete rules as illustrated in this work. Moreover, it would be interesting to further tweak the counterexample generation in order to yield particularly illustrative voting situations, e.g., minimal ones or scenarios of some interesting structure. A more ambitious endeavor would be to implement our contribution with an argumentation framework, where a user could agree or disagree with particular counterexamples, in order to automatically adjust the desired voting rule in order to recommend some kind of optimum.

For our method for automatic margin computation, it would be interesting to integrate further desired constraints and queries, in order to obtain margins leading to a particularly interesting election outcome. Moreover, it would be worthwhile to adapt our approach to more complex constraints in order to compute, e.g., some kind of quantitative distances for quantitative fairness properties.

Finally, for our case study on the credential generation software, it remains to close the formal gap in the justification of the employed ideal functionality.

References

Abe, Yuta, Yu-ichi Hayashi, Takaaki Mizuki, and Hideaki Sone (2018). "Five-Card AND Protocol in Committed Format Using Only Practical Shuffles." In: *5th Workshop on ASIA Public-Key Cryptography (APKC@AsiaCCS)* (Incheon, Republic of Korea, June 4, 2018). Ed. by Keita Emura, Jae Hong Seo, and Yohei Watanabe. Association for Computing Machinery, pp. 3–8. DOI: 10.1145/3197507.3197510.

Adida, Ben (2008). "Helios: Web-based Open-Audit Voting." In: *17th USENIX Security Symposium* (San Jose, CA, USA, July 28–Aug. 1, 2008). Ed. by Paul C. van Oorschot. USENIX Association, pp. 335–348. URL: http://www.usenix.org/events/sec08/tech/full_papers/adida/adida.pdf.

Ahrendt, Wolfgang, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich (2016). *Deductive Software Verification - The KeY Book: From Theory to Practice*. Vol. 10001. Lecture Notes in Computer Science. Springer. DOI: 10.1007/978-3-319-49812-6.

Apt, Krzysztof R. (1997). *From Logic Programming to Prolog*. Prentice Hall International Series in Computer Science. Upper Saddle River, NJ, United States: Prentice Hall.

Arrow, Kenneth Joseph (1950). "A Difficulty in the Concept of Social Welfare." Trans. by University of Chicago Press. *Journal of Political Economy* 58.4, pp. 328–346. DOI: 10.2307/1828886.

– (1951). *Social Choice and Individual Values*. Monographs / Cowles Commission for Research in Economics 12. New York, USA: Wiley.

Ashur, Tomer, Orr Dunkelman, and Nimrod Talmon (2016). "Breaching the Privacy of Israel's Paper Ballot Voting System." In: *First International Joint Conference on Electronic Voting (E-Vote-ID 2016)* (Bregenz, Austria, Oct. 18–21, 2016). Ed. by Robert Krimmer, Melanie Volkamer, Jordi Barrat, Josh Benaloh, Nicole J. Goodman, Peter Y. A. Ryan, and Vanessa Teague. Vol. 10141. Lecture Notes in Computer Science. Springer, pp. 108–124. DOI: 10.1007/978-3-319-52240-1_7.

Avalle, Matteo, Alfredo Pironti, and Riccardo Sisto (2014). "Formal verification of security protocol implementations: a survey." *Formal Aspects of Computing* 26.1, pp. 99–123. DOI: 10.1007/s00165-012-0269-9.

Baldwin, Joseph M. (1926). "The Technique of the Nanson Preferential Majority System of Election." *Proceedings of the Royal Society of Victoria* 39, pp. 42–52.

Balinski, Michel L. and H. Peyton Young (2010). *Fair Representation: Meeting the Ideal of One Man, One Vote*. Brookings Institution Press. URL: http://www.jstor.org/stable/10.7864/j.ctvcb59f6.

Banerjee, Anindya and David A. Naumann (2005). "Stack-based access control and secure information flow." *The Journal of Functional Programming* 15.2, pp. 131–177. DOI: 10.1017/S0956796804005453.

Barrett, Clark W. and Cesare Tinelli (2018). "Satisfiability Modulo Theories." In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Springer, pp. 305–343. DOI: 10.1007/978-3-319-10575-8_11.

Barthe, Gilles, Juan Manuel Crespo, and César Kunz (2011). "Relational Verification Using Product Programs." In: *17th International Symposium on Formal Methods (FM 2011)* (Limerick, Ireland, June 20–24, 2011). Ed. by Michael Butler and Wolfram Schulte. Vol. 6664. Lecture Notes in Computer Science. Springer, pp. 200–214. DOI: 10.1007/978-3-642-21437-0_17.

Barthe, Gilles, Pedro R. D'Argenio, and Tamara Rezk (2004). "Secure Information Flow by Self-Composition." In: *17th Computer Security Foundations Workshop (CSFW-17 2004)*. IEEE Computer Society, pp. 100–114. DOI: 10.1109/CSFW.2004.17.

Barthe, Gilles, David Pichardie, and Tamara Rezk (2013). "A certified lightweight non-interference Java bytecode verifier." *Mathematical Structures in Computer Science* 23.5, pp. 1032–1081. DOI: 10.1017/S0960129512000850.

Bartholdi, John J. and James B. Orlin (1991). "Single Transferable Vote Resists Strategic Voting." *Social Choice and Welfare* 8, pp. 341–354. DOI: 10.1007/BF00183045.

Beckert, Bernhard, Simon Bischof, Mihai Herda, Michael Kirsten, and Marko Kleine Büning (2018). "Using Theorem Provers to Increase the Precision of Dependence Analysis for Information Flow Control." In: *20th International Conference on Formal Engineering Methods - Formal Methods and Software Engineering (ICFEM 2018)* (Gold Coast, Australia, Nov. 12–16, 2018). Ed. by Jing Sun and Meng Sun. Vol. 11232. Lecture Notes in Computer Science. Springer, pp. 284–300. DOI: 10.1007/978-3-030-02450-5_17.

Beckert, Bernhard, Thorsten Bormer, Rajeev Goré, Michael Kirsten, and Carsten Schürmann (2017). "An Introduction to Voting Rule Verification." In: *Trends in Computational Social Choice*. Ed. by Ulle Endriss. .II: Techniques. AI Access. Chap. 14, pp. 269–287. URL: http://research.illc.uva.nl/COST-IC1205/Book/.

Beckert, Bernhard, Thorsten Bormer, Michael Kirsten, Till Neuber, and Mattias Ulbrich (2016). "Automated Verification for Functional and Relational Properties of Voting Rules." In: *Sixth International Workshop on Computational Social Choice (COMSOC 2016)*

(Toulouse, France, June 22–24, 2016). Ed. by Umberto Grandi and Jeffrey S. Rosenschein. URL: https://irit.fr/COMSOC-2016/proceedings/BeckertEtAlCOMSOC2016.pdf.

Beckert, Bernhard, Thorsten Bormer, Florian Merz, and Carsten Sinz (2012). "Integration of Bounded Model Checking and Deductive Verification." In: *International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2011), Revised Selected Papers* (Turin, Italy, Oct. 5–7, 2011). Ed. by Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov. Vol. 7421. Lecture Notes in Computer Science. Springer, pp. 86–104. DOI: 10.1007/978-3-642-31762-0_7.

Beckert, Bernhard, Achim Brelle, Rüdiger Grimm, Nicolas Huber, Michael Kirsten, Ralf Küsters, Jörn Müller-Quade, Maximilian Noppel, Kai Reinhard, Jonas Schwab, Rebecca Schwerdt, Tomasz Truderung, Melanie Volkamer, and Cornelia Winter (2019). "GI Elections with POLYAS: a Road to End-to-End Verifiable Elections." In: *Fourth International Joint Conference on Electronic Voting (E-Vote-ID 2019)* (Lochau / Bregenz, Austria, Oct. 1–4, 2019). Ed. by Robert Krimmer, Melanie Volkamer, Bernhard Beckert, Véronique Cortier, Ardita Driza-Maurer, David Duenas-Cid, Jörg Helbach, Reto Koenig, Iuliia Krivonosova, Ralf Küsters, Peter Rønne, Uwe Serdült, and Oliver Spycher. Proceedings E-Vote-ID 2019. TalTech Press, pp. 293–294. URL: https://digi.lib.ttu.ee/i/?13563.

Beckert, Bernhard, Daniel Bruns, Vladimir Klebanov, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich (2013). "Information Flow in Object-Oriented Software." In: *23rd International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2013), Revised Selected Papers* (Madrid, Spain, Sept. 18–19, 2013). Ed. by Gopal Gupta and Ricardo Peña. Vol. 8901. Lecture Notes in Computer Science. Springer, pp. 19–37. DOI: 10.1007/978-3-319-14125-1_2.

Beckert, Bernhard, Rajeev Goré, and Carsten Schürmann (2013). "Analysing Vote Counting Algorithms via Logic - And its Application to the CADE Election System." In: *24th International Conference on Automated Deduction (CADE-24)* (Lake Placid, NY, USA, June 9–14, 2013). Ed. by Maria Paola Bonacina. Vol. 7898. Lecture Notes in Computer Science. Springer, pp. 135–144. DOI: 10.1007/978-3-642-38574-2_9.

Beckert, Bernhard, Rajeev Goré, Carsten Schürmann, Thorsten Bormer, and Jian Wang (2014). "Verifying Voting Schemes." *Journal of Information Security and Applications* 19.2, pp. 115–129. DOI: 10.1016/j.jisa.2014.04.005.

Beckert, Bernhard, Mihai Herda, Michael Kirsten, and Shmuel Tyszberowicz (2020). "Integration of Static and Dynamic Analysis Techniques for Checking Noninterference." In: *Deductive Software Verification: Future Perspectives - Reflections on the Occasion of 20 Years of KeY*. Ed. by Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, and Mattias Ulbrich. Vol. 12345.V: Integration of Verification Techniques. Lecture Notes in Computer Science. Springer. Chap. 12, pp. 287–312. DOI: 10.1007/978-3-030-64354-6_12.

Beckert, Bernhard, Michael Kirsten, Jonas Klamroth, and Mattias Ulbrich (2020). "Modular Verification of JML Contracts Using Bounded Model Checking." In: *9th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2020)* (Rhodes, Greece, Oct. 26–30, 2020). Ed. by Tiziana Margaria and Bernhard Stef-

fen. Vol. 12476.I: Verification Principles. Lecture Notes in Computer Science. Springer, pp. 60–80. DOI: 10.1007/978-3-030-61362-4_4.

Beckert, Bernhard, Michael Kirsten, Vladimir Klebanov, and Carsten Schürmann (2017). "Automatic Margin Computation for Risk-Limiting Audits." In: *First International Joint Conference on Electronic Voting – formerly known as EVOTE and VoteID (E-Vote-ID 2016)* (Lochau / Bregenz, Austria, Oct. 18–21, 2017). Ed. by Robert Krimmer, Melanie Volkamer, Jordi Barrat, Josh Benaloh, Nicole J. Goodman, Peter Y. A. Ryan, and Vanessa Teague. Vol. 10141. Lecture Notes in Computer Science. Springer, pp. 18–35. DOI: 10.1007/978-3-319-52240-1_2.

Beckert, Bernhard, Vladimir Klebanov, and Benjamin Weiß (2016). "Dynamic Logic for Java." In: *Deductive Software Verification - The KeY Book: From Theory to Practice*. Ed. by Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich. Vol. 10001.I: Foundations. Lecture Notes in Computer Science. Springer. Chap. 3, pp. 49–106. DOI: 10.1007/978-3-319-49812-6_3.

Bernhard, David and Bogdan Warinschi (2014). "Cryptographic Voting — A Gentle Introduction." In: *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*. Ed. by Alessandro Aldini, Javier López, and Fabio Martinelli. Vol. 8604. Lecture Notes in Computer Science. Springer, pp. 167–211. DOI: 10.1007/978-3-319-10082-1_7.

Bernhard, Matthew, Josh Benaloh, John Alex Halderman, Ronald L. Rivest, Peter Y. A. Ryan, Philip B. Stark, Vanessa Teague, Poorvi L. Vora, and Dan S. Wallach (2017). "Public Evidence from Secret Ballots." In: *Second International Joint Conference on Electronic Voting (E-Vote-ID 2017)* (Bregenz, Austria, Oct. 24–27, 2017). Ed. by Robert Krimmer, Melanie Volkamer, Nadja Braun Binder, Norbert Kersting, Olivier Pereira, and Carsten Schürmann. Vol. 10615. Lecture Notes in Computer Science. Springer, pp. 84–109. DOI: 10.1007/978-3-319-68687-5_6.

Biere, Armin and Daniel Kröning (2018). "SAT-Based Model Checking." In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem. Springer, pp. 277–303. DOI: 10.1007/978-3-319-10575-8_10.

Black, Duncan (1958). *The Theory of Committees and Elections*. Vol. 36. 137. Cambridge University Press. DOI: 10.1017/S0031819100058204.

Blanchet, Bruno (2012). "Security Protocol Verification: Symbolic and Computational Models." In: *First International Conference on Principles of Security and Trust (POST 2012), held as Part of ETAPS 2012: the European Joint Conferences on Theory and Practice of Software* (Tallinn, Estonia, Mar. 24–Apr. 1, 2012). Ed. by Pierpaolo Degano and Joshua D. Guttman. Vol. 7215. Lecture Notes in Computer Science. Springer, pp. 3–29. DOI: 10.1007/978-3-642-28641-4_2.

Blom, Michelle L., Vanessa Teague, Peter J. Stuckey, and Ron Tidhar (2016). "Efficient Computation of Exact IRV Margins." In: *22nd European Conference on Artificial Intelligence (ECAI 2016) Including Prestigious Applications of Artificial Intelligence (PAIS 2016)* (The Hague, The Netherlands, Aug. 29–Sept. 2, 2016). Ed. by Gal A. Kaminka, Maria Fox,

Paolo Bouquet, Eyke Hüllermeier, Virginia Dignum, Frank Dignum, and Frank van Harmelen. Vol. 285. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 480–488. DOI: 10.3233/978-1-61499-672-9-480.

Bohr, Stephan (2020). "Formal Verification of Condorcet Voting Rules Using Composable Modules." Bachelor's Thesis. ITI Beckert, Karlsruhe Institute of Technology (KIT).

Brams, Steven J. and Peter C. Fishburn (1988). "Does Approval Voting Elect the Lowest Common Denominator?" *PS: Political Science & Politics* 21.02, pp. 277–284.

Brandt, Felix, Vincent Conitzer, Ulle Endriss, Jérôme Lang, and Ariel D. Procaccia (2016). *Handbook of Computational Social Choice*. Cambridge University Press. DOI: 10.1017/CBO9781107446984.

Brandt, Felix and Christian Geist (2016). "Finding Strategyproof Social Choice Functions via SAT Solving." *Journal of Artificial Intelligence Research* 55, pp. 565–602. DOI: 10.1613/jair.4959.

Brandt, Felix, Christian Geist, and Dominik Peters (2017). "Optimal bounds for the no-show paradox via SAT solving." *Math. Soc. Sci.* 90, pp. 18–27. DOI: 10.1016/j.mathsocsci.2016.09.003.

Bryant, Randal E., Daniel Kroening, Joël Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan A. Brady (2007). "Deciding Bit-Vector Arithmetic with Abstraction." In: *13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007) held as part of ETAPS 2007: the Joint European Conferences on Theory and Practice of Software* (Braga, Portugal, Mar. 24–Apr. 1, 2007). Ed. by Orna Grumberg and Michael Huth. Vol. 4424. Lecture Notes in Computer Science. Springer, pp. 358–372. DOI: 10.1007/978-3-540-71209-1_28.

Bundestagsdrucksache 17/11819 (2012). *Entwurf eines Zweiundzwanzigsten Gesetzes zur Änderung des Bundeswahlgesetzes*. Bundesanzeiger Verlagsgesellschaft mbH. URL: http://dipbt.bundestag.de/dip21/btd/17/118/1711819.pdf.

Cadoli, Marco and Toni Mancini (2007). "Using a Theorem Prover for Reasoning on Constraint Problems." *Applied Artificial Intelligence* 21.4&5, pp. 383–404. DOI: 10.1080/08839510701252650.

Cailloux, Olivier and Ulle Endriss (2014). "Eliciting a Suitable Voting Rule via Examples." In: *21st European Conference on Artificial Intelligence (ECAI 2014), Including Prestigious Applications of Intelligent Systems (PAIS 2014)* (Prague, Czech Republic, Aug. 18–22, 2014). Ed. by Torsten Schaub, Gerhard Friedrich, and Barry O'Sullivan. Vol. 263. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 183–188. DOI: 10.3233/978-1-61499-419-0-183.

– (2016). "Arguing about Voting Rules." In: *International Conference on Autonomous Agents & Multiagent Systems (AAMAS '16)* (Singapore, Singapore, May 9–13, 2016). Ed. by Catholijn M. Jonker, Stacy Marsella, John Thangarajah, and Karl Tuyls. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), pp. 287–295. URL: http://dl.acm.org/citation.cfm?id=2936968.

Cailloux, Olivier and Yves Meinard (2020). "A formal framework for deliberated judgment." *Theory and Decision* 88, pp. 269–295. DOI: 10.1007/s11238-019-09722-7.

Cary, David (2011). "Estimating the Margin of Victory for Instant-Runoff Voting." In: *Conference on Electronic Voting Technology/Workshop on Trustworthy Elections (EVT/WOTE '11)* (San Francisco, CA, USA, Aug. 8–9, 2011). Ed. by Hovav Shacham and Vanessa Teague. USENIX Association. URL: https://www.usenix.org/conference/evtwote-11/estimating-margin-victory-instant-runoff-voting.

Charwat, Günther and Andreas Pfandler (2015). "Democratix: A Declarative Approach to Winner Determination." In: *4th International Conference on Algorithmic Decision Theory (ADT 2015)* (Lexington, KY, USA, Sept. 27–30, 2015). Ed. by Toby Walsh. Vol. 9346. Lecture Notes in Computer Science. Springer, pp. 253–269. DOI: 10.1007/978-3-319-23114-3_16.

Chatterjee, Siddharth and Arunava Sen (2014). "Automated Reasoning in Social Choice Theory: Some Remarks." *Mathematics in Computer Science* 8.1, pp. 5–10. DOI: 10.1007/s11786-014-0177-x.

Clarke, Edmund M., Daniel Kroening, and Flavio Lerda (2004). "A Tool for Checking ANSI-C Programs." In: *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004), held as part of ETAPS 2004: the Joint European Conferences on Theory and Practice of Software* (Barcelona, Spain, Mar. 29–Apr. 2, 2004). Ed. by Kurt Jensen and Andreas Podelski. Vol. 2988. Lecture Notes in Computer Science. Springer, pp. 168–176. DOI: 10.1007/978-3-540-24730-2_15.

Clarke, Edmund M., Daniel Kroening, and Karen Yorav (2003). "Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking." In: *The 40th Design Automation Conference (DAC 2003)* (Anaheim, CA, USA, June 2–6, 2003). Association for Computing Machinery, pp. 368–371. DOI: 10.1145/775832.775928.

Clarkson, Michael E., Stephen Chong, and Andrew C. Myers (2007). "Civitas: A Secure Remote Voting System." In: *Frontiers of Electronic Voting* (Wadern, Germany, July 29–Aug. 3, 2007). Ed. by David Chaum, Miroslaw Kutylowski, Ronald L. Rivest, and Peter Y. A. Ryan. Vol. 07311. Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany. URL: http://drops.dagstuhl.de/opus/volltexte/2008/1296.

Colmerauer, Alain and Philippe Roussel (1993). "The Birth of Prolog." In: *The Second Conference on History of Programming Languages (HOPL-II)* (Cambridge, Massachusetts, USA, Apr. 20–23, 1993). Ed. by John A. N. Lee and Jean E. Sammet. Association for Computing Machinery, pp. 37–52. DOI: 10.1145/154766.155362.

Colomer, Josep M. (2013). "Ramon Llull: from 'Ars electionis' to social choice theory." *Social Choice and Welfare* 40.2, pp. 317–328. DOI: 10.1007/s00355-011-0598-2.

Conitzer, Vincent and Tuomas Sandholm (2012). "Common Voting Rules as Maximum Likelihood Estimators." *Computing Research Repository (CoRR)*. ID: abs/1207.1368.

Copeland, Arthur H. (1951). "A 'reasonable' social welfare function. Notes from a seminar on applications of mathematics to the social sciences." In: *University of Michigan*. Mimeographed Notes. Ann Arbor, MI, US.

Cornelio, Cristina, Michele Donini, Andrea Loreggia, Maria Silvia Pini, and Francesca Rossi (2021). "Voting with random classifiers (VORACE): theoretical and experimental analysis." *Autonomous Agents and Multi-Agent Systems* 35.22. DOI: 10.1007/s10458-021-09504-y.

Crawford, James M., Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy (1996). "Symmetry-Breaking Predicates for Search Problems." In: *Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96)* (Cambridge, Massachusetts, USA, Nov. 5–8, 1996). Ed. by Luigia Carlucci Aiello, Jon Doyle, and Stuart C. Shapiro. Morgan Kaufmann Publishers Inc., pp. 148–159. URL: https://dl.acm.org/doi/abs/10.5555/3087368.3087386.

Crépeau, Claude and Joe Kilian (1993). "Discreet Solitary Games." In: *13th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO '93)* (Santa Barbara, California, USA, Aug. 22–26, 1993). Ed. by Douglas R. Stinson. Vol. 773. Lecture Notes in Computer Science. Springer, pp. 319–330. DOI: 10.1007/3-540-48329-2_27.

Danmarks Statistik (2015). *Befolkning og Valg*. Online, accessed 13-December-2021. URL: http://www.dst.dk/valg/Valg1487635/other/2015-Folketingsvalg.pdf.

Darga, Paul T., Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov (2004). "Exploiting Structure in Symmetry Detection for CNF." In: *41st Annual Design Automation Conference (DAC 2004)* (San Diego, CA, USA, June 7–11, 2004). Ed. by Sharad Malik, Limor Fix, and Andrew B. Kahng. Association for Computing Machinery, pp. 530–534. DOI: 10.1145/996566.996712.

Darvas, Ádám, Reiner Hähnle, and David Sands (2005). "A Theorem Proving Approach to Analysis of Secure Information Flow." In: *Second International Conference on Security in Pervasive Computing (SPC 2005)* (Boppard, Germany, Apr. 6, 2005–Apr. 8, 2004). Ed. by Dieter Hutter and Markus Ullmann. Vol. 3450. Lecture Notes in Computer Science. Springer, pp. 193–209. DOI: 10.1007/978-3-540-32004-3_20.

Dawson, Jeremy E., Rajeev Goré, and Thomas Meumann (2015). "Machine-Checked Reasoning About Complex Voting Schemes Using Higher-Order Logic." In: *5th International Conference on E-Voting and Identity (VoteID 2015)* (Bern, Switzerland, Sept. 2–4, 2015). Ed. by Rolf Haenni, Reto E. Koenig, and Douglas Wikström. Vol. 9269. Lecture Notes in Computer Science. Springer, pp. 142–158. DOI: 10.1007/978-3-319-22270-7_9.

den Boer, Bert (1989). "More Efficient Match-Making and Satisfiability: *The Five Card Trick*." In: *Workshop on the Theory and Application of Cryptographic Techniques (Advances in Cryptology — EUROCRYPT '89)* (Houthalen, Belgium, Apr. 10–13, 1989). Ed. by Jean-Jacques Quisquater and Joos Vandewalle. Vol. 434. Lecture Notes in Computer Science. Springer, pp. 208–217. DOI: 10.1007/3-540-46885-4_23.

Denning, Dorothy E. (1976). "A Lattice Model of Secure Information Flow." *Communication of the ACM* 19.5, pp. 236–243. DOI: 10.1145/360051.360056.

Denning, Dorothy E. and Peter J. Denning (1977). "Certification of Programs for Secure Information Flow." *Communication of the ACM* 20.7, pp. 504–513. DOI: 10.1145/359636.359712.

Dennis, Greg, Kuat Yessenov, and Daniel Jackson (2008). "Bounded Verification of Voting Software." In: *Second International Conference on Verified Software: Theories, Tools, Experiments (VSTTE 2008)* (Toronto, Canada, Oct. 6–9, 2008). Ed. by Natarajan Shankar and Jim Woodcock. Vol. 5295. Lecture Notes in Computer Science. Springer, pp. 130–145. DOI: 10.1007/978-3-540-87873-5_13.

Diekhoff, Karsten, Michael Kirsten, and Jonas Krämer (2019). "Formal Property-Oriented Design of Voting Rules Using Composable Modules." In: *6th International Conference on Algorithmic Decision Theory (ADT 2019)* (Durham, NC, USA, Oct. 10–27, 2019). Ed. by Saša Pekeč and Kristen Brent Venable. Vol. 11834.Short Papers. Lecture Notes in Artificial Intelligence. Springer, pp. 164–166. DOI: 10.1007/978-3-030-31489-7.

– (2020). "Verified Construction of Fair Voting Rules." In: *29th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2019), Revised Selected Papers* (Porto, Portugal, Oct. 8–10, 2019). Ed. by Maurizio Gabbrielli. Vol. 12042. Lecture Notes in Computer Science. Springer, pp. 90–104. DOI: 10.1007/978-3-030-45260-5_6.

Dijkstra, Edsger W. (1975). "Guarded Commands, Nondeterminacy and Formal Derivation of Programs." *Communications of the ACM* 18.8, pp. 453–457. DOI: 10.1145/360933.360975.

Do, Quoc Huy, Richard Bubel, and Reiner Hähnle (2017). "Automatic detection and demonstrator generation for information flow leaks in object-oriented programs." *Computers & Security* 67, pp. 335–349. DOI: 10.1016/j.cose.2016.12.002.

Eén, Niklas and Niklas Sörensson (2003). "An Extensible SAT-Solver." In: *6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), Selected Revised Papers* (Santa Margherita Ligure, Italy, May 5–8, 2003). Ed. by Enrico Giunchiglia and Armando Tacchella. Vol. 2919. Lecture Notes in Computer Science. Springer, pp. 502–518. DOI: 10.1007/978-3-540-24605-3_37.

Elkind, Edith, Piotr Faliszewski, and Arkadii Slinko (2015). "Distance rationalization of voting rules." *Social Choice and Welfare* 45.2, pp. 345–377. DOI: 10.1007/s00355-015-0892-5.

Elklit, Jørgen, Anne Birte Pade, and Nicoline Nyholm Miller (2011). *The Parliamentary Electoral System in Denmark*. Online, accessed 13-December-2021. Ministry of the Interior, Health, and The Danish Parliament. URL: https://www.thedanishparliament.dk/-/media/pdf/publikationer/english/the-parliamentary-system-of-denmark_2011.ashx.

Endriss, Ulle (2017). *Trends in Computational Social Choice*. AI Access. URL: https://research.illc.uva.nl/COST-IC1205/BookDocs/TrendsCOMSOC.pdf.

European Commission for Democracy through Law (Venice Commission) (Oct. 25, 2018). *Code of Good Practice in Electoral Matters. Guidelines and Explanatory Report*. Opinion 190/2002. Version 52. Strasbourg, France: Council of Europe. URL: https://www.venice.coe.int/webforms/documents/default.aspx?pdffile=CDL-AD(2002)023rev2-cor-e.

Felsing, Dennis, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich (2014). "Automating regression verification." In: *International Conference on Automated Software Engineering (ASE '14)* (Vasteras, Sweden, Sept. 15–19, 2014). Ed. by Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher. Association for Computing Machinery / IEEE Computer Society, pp. 349–360. DOI: 10.1145/2642937.2642987.

Filliâtre, Jean-Christophe (2011). "Deductive software verification." *International Journal on Software Tools for Technology Transfer* 13.5, pp. 397–403. DOI: 10.1007/s10009-011-0211-0.

Fishburn, Peter C. (1973). *The Theory of Social Choice*. Vol. 1757. Princeton Legacy Library. Princeton University Press. DOI: 10.1515/9781400868339.

Fournet, Cédric, Markulf Kohlweiss, and Pierre-Yves Strub (2011). "Modular code-based cryptographic verification." In: *18th Conference on Computer and Communications Security (CCS 2011)* (Chicago, Illinois, USA, Oct. 17–21, 2011). Ed. by Yan Chen, George Danezis, and Vitaly Shmatikov. Association for Computing Machinery, pp. 341–350. DOI: 10.1145/2046707.2046746.

Franz, Martin, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith (2014). "CBMC-GC: An ANSI C Compiler for Secure Two-Party Computations." In: *23rd International Conference on Compiler Construction (CC 2014), held as Part of ETAPS 2014: the European Joint Conferences on Theory and Practice of Software* (Grenoble, France, Apr. 5–13, 2014). Ed. by Albert Cohen. Vol. 8409. Lecture Notes in Computer Science. Springer, pp. 244–249. DOI: 10.1007/978-3-642-54807-9_15.

Gallagher, Michael (1991). "Proportionality, Disproportionality and Electoral Systems." *Electoral Studies* 10.1, pp. 33–51. DOI: 10.1016/0261-3794(91)90004-C.

– (2013). "Monotonicity and Non-Monotonicity at PR-STV Elections." In: *Annual Conference on Elections, Public Opinion and Parties (EPOP 2013)* (Lancaster, UK, Sept. 13–15, 2013). Lancaster University.

Geist, Christian and Ulle Endriss (2011). "Automated Search for Impossibility Theorems in Social Choice Theory: Ranking Sets of Objects." *Journal of Artificial Intelligence Research* 40, pp. 143–174. DOI: 10.1613/jair.3126.

Ghale, Milad K., Rajeev Goré, Dirk Pattinson, and Mukesh Tiwari (2018). "Modular Formalisation and Verification of STV Algorithms." In: *Third International Joint Conference on Electronic Voting (E-Vote-ID 2018)* (Bregenz, Austria, Oct. 2–5, 2018). Ed. by Robert Krimmer, Melanie Volkamer, Véronique Cortier, Rajeev Goré, Manik Hapsara, Uwe Serdült, and David Duenas-Cid. Vol. 11143. Lecture Notes in Computer Science. Springer, pp. 51–66. DOI: 10.1007/978-3-030-00419-4_4.

Gibbard, Allan (1973). "Manipulation of Voting Schemes: A General Result." *Econometrica* 41.4, pp. 587–601. DOI: 10.2307/1914083.

Goguen, Joseph A. and José Meseguer (1982). "Security Policies and Security Models." In: *Symposium on Security and Privacy (SP)* (Oakland, CA, USA, Apr. 26–28, 1982). IEEE Computer Society, pp. 11–20. DOI: 10.1109/SP.1982.10014.

Gomes, Carla P., Henry A. Kautz, Ashish Sabharwal, and Bart Selman (2008). "Satisfiability Solvers." In: *Handbook of Knowledge Representation*. Ed. by Frank van Harmelen,

Vladimir Lifschitz, and Bruce W. Porter. Vol. 3. Foundations of Artificial Intelligence. Elsevier, pp. 89–134. DOI: 10.1016/S1574-6526(07)03002-7.

Goré, Rajeev and Thomas Meumann (2014). "Proving the Monotonicity Criterion for a Plurality Vote-Counting Program as a Step Towards Verified Vote-Counting." In: *6th International Conference on Electronic Voting: Verifying the Vote (EVOTE 2014)* (Lochau / Bregenz, Austria, Oct. 29–31, 2014). Ed. by Robert Krimmer and Melanie Volkamer. IEEE Computer Society, pp. 1–7. DOI: 10.1109/EVOTE.2014.7001138.

Graf, Jürgen, Martin Hecker, and Martin Mohr (2013). "Using JOANA for Information Flow Control in Java Programs - A Practical Guide." In: *Software Engineering 2013 - Workshopband (inkl. Doktorandensymposium), Fachtagung des GI-Fachbereichs Softwaretechnik* (Aachen, Germany, Feb. 26–Mar. 1, 2013). Ed. by Stefan Wagner and Horst Lichter. Vol. 215. Lecture Notes in Informatics. Gesellschaft für Informatik (GI), pp. 123–138. URL: https://dl.gi.de/20.500.12116/17361.

Grilli di Cortona, Pietro, Cecilia Manzi, Aline Pennisi, Federica Ricca, and Bruno Simeone (1999). *Evaluation and optimization of electoral systems*. Society for Industrial and Applied Mathematics. DOI: 10.1137/1.9780898719819.

Haines, Thomas (2019). "A Description and Proof of a Generalised and Optimised Variant of Wikström's Mixnet." *Computing Research Repository (CoRR)*. ID: abs/1901.08371.

Hammer, Christian and Gregor Snelting (2009). "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs." *International Journal of Information Security* 8.6, pp. 399–422. DOI: 10.1007/s10207-009-0086-1.

Hoare, Charles Antony Richard (1969). "An Axiomatic Basis for Computer Programming." *Communications of the ACM* 12.10, pp. 576–580. DOI: 10.1145/363235.363259.

Holzer, Andreas, Christian Schallhart, Michael Tautschnig, and Helmut Veith (2008). "FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement." In: *20th International Conference on Computer Aided Verification (CAV 2008)* (Princeton, NJ, USA, July 7–14, 2008). Ed. by Aarti Gupta and Sharad Malik. Vol. 5123. Lecture Notes in Computer Science. Springer, pp. 209–213. DOI: 10.1007/978-3-540-70545-1_20.

Kastner, Julia, Alexander Koch, Stefan Walzer, Daiki Miyahara, Yu-ichi Hayashi, Takaaki Mizuki, and Hideaki Sone (2017). "The Minimum Number of Cards in Practical Card-Based Protocols." In: *Advances in Cryptology - 23rd International Conference on the Theory and Applications of Cryptology and Information Security (ASIACRYPT 2017)* (Hong Kong, China, Dec. 3–7, 2017). Ed. by Tsuyoshi Takagi and Thomas Peyrin. Vol. 10626.III. Lecture Notes in Computer Science. Springer, pp. 126–155. DOI: 10.1007/978-3-319-70700-6_5.

Kirsten, Michael and Olivier Cailloux (2018). "Towards automatic argumentation about voting rules." In: *4ème Conférence Nationale sur les Applications Pratiques de l'Intelligence Artificielle (APIA 2018)* (Nancy, France, July 2–6, 2018). Ed. by Sandra Bringay and Juliette Mattioli. URL: https://hal.archives-ouvertes.fr/hal-01830911.

Koch, Alexander (2018). "The Landscape of Optimal Card-based Protocols." *IACR Cryptology ePrint Archive*, p. 951. URL: https://eprint.iacr.org/2018/951.

– (2019). "Cryptographic Protocols from Physical Assumptions." PhD thesis. Karlsruhe: Karlsruhe Institute of Technology (KIT). DOI: 10.5445/IR/1000097756.

Koch, Alexander, Michael Schrempp, and Michael Kirsten (2019). "Card-Based Cryptography Meets Formal Verification." In: *25th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT 2019)* (Kobe, Japan, Dec. 8–12, 2019). Ed. by Steven D. Galbraith and Shiho Moriai. Vol. 11921.I. Lecture Notes in Computer Science. Springer, pp. 488–517. DOI: 10.1007/978-3-030-34578-5_18.

– (2021). "Card-Based Cryptography Meets Formal Verification." Trans. by Takaaki Mizuki. *New Generation Computing* 39.1: *Special Issue on Card-Based Cryptography*, pp. 115–158. DOI: 10.1007/s00354-020-00120-0.

Koch, Alexander, Stefan Walzer, and Kevin Härtel (2015). "Card-based Cryptographic Protocols Using a Minimal Number of Cards." In: *21st International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT 2015)* (Auckland, New Zealand, Nov. 29–Dec. 3, 2015). Ed. by Tetsu Iwata and Jung Hee Cheon. Vol. 9452.I. Lecture Notes in Computer Science. Springer, pp. 783–807. DOI: 10.1007/978-3-662-48797-6_32.

Koitmäe, Arne, Jan Willemson, and Priit Vinkel (2021). "Vote Secrecy and Voter Feedback in Remote Voting - Can We Have Both?" In: *6th International Joint Conference on Electronic Voting (E-Vote-ID 2021)* (Virtual Event, Oct. 5–8, 2021). Ed. by Robert Krimmer, Melanie Volkamer, David Duenas-Cid, Oksana Kulyk, Peter B. Rønne, Mihkel Solvak, and Micha Germann. Vol. 12900. Lecture Notes in Computer Science. Springer, pp. 140–154. DOI: 10.1007/978-3-030-86942-7_10.

Küsters, Ralf and Johannes Müller (2017). "Cryptographic Security Analysis of E-voting Systems: Achievements, Misconceptions, and Limitations." In: *Second International Joint Conference on Electronic Voting (E-Vote-ID 2017)* (Bregenz, Austria, Oct. 24–27, 2017). Ed. by Robert Krimmer, Melanie Volkamer, Nadja Braun Binder, Norbert Kersting, Olivier Pereira, and Carsten Schürmann. Vol. 10615. Lecture Notes in Computer Science. Springer, pp. 21–41. DOI: 10.1007/978-3-319-68687-5_2.

Küsters, Ralf, Tomasz Truderung, Bernhard Beckert, Daniel Bruns, Michael Kirsten, and Martin Mohr (2015). "A Hybrid Approach for Proving Noninterference of Java Programs." In: *28th Computer Security Foundations Symposium (CSF 2015)* (Verona, Italy, July 13–17, 2015). Ed. by Cédric Fournet and Michael Hicks. IEEE Computer Society, pp. 305–319. DOI: 10.1109/CSF.2015.28.

Küsters, Ralf, Tomasz Truderung, and Juergen Graf (2012). "A Framework for the Cryptographic Verification of Java-Like Programs." In: *25th Computer Security Foundations Symposium (CSF 2012)* (Cambridge, MA, USA, June 25–27, 2012). Ed. by Stephen Chong. IEEE Computer Society, pp. 198–212. DOI: 10.1109/CSF.2012.9.

Leavens, Gary T., Albert L. Baker, and Clyde Ruby (2006). "Preliminary Design of JML: A Behavioral Interface Specification Language for Java." *ACM SIGSOFT Software Engineering Notes* 31.3, pp. 1–38. DOI: 10.1145/1127878.1127884.

Lindeman, Mark and Philip B. Stark (2012). "A Gentle Introduction to Risk-Limiting Audits." *IEEE Security & Privacy* 10.5, pp. 42–49. DOI: 10.1109/MSP.2012.56.

Lundin, David (2010). "Component Based Electronic Voting Systems." In: *Towards Trustworthy Elections, New Directions in Electronic Voting*. Ed. by David Chaum, Markus Jakobsson, Ronald L. Rivest, Peter Y. A. Ryan, Josh Benaloh, Miroslaw Kutylowski, and Ben Adida. Vol. 6000. Lecture Notes in Computer Science. Springer, pp. 260–273. DOI: 10.1007/978-3-642-12980-3_16.

Maggs, Bruce M. and Ramesh K. Sitaraman (2015). "Algorithmic Nuggets in Content Delivery." *ACM SIGCOMM Computer Communication Review* 45.3, pp. 52–66. DOI: 10.1145/2805789.2805800.

Magrino, Thomas R., Ronald L. Rivest, Emily Shen, and David Wagner (2011). "Computing the Margin of Victory in IRV Elections." In: *Workshop on Electronic Voting Technology / Workshop on Trustworthy Elections (EVT/WOTE '11)* (San Francisco, CA, USA, Aug. 8–9, 2011). Ed. by Hovav Shacham and Vanessa Teague. USENIX Association. URL: https://www.usenix.org/conference/evtwote-11/computing-margin-victory-irv-elections.

Mancini, Toni and Marco Cadoli (2005). "Detecting and Breaking Symmetries by Reasoning on Problem Specifications." In: *6th International Symposium on Abstraction, Reformulation and Approximation (SARA 2005)* (Airth Castle, Scotland, UK, July 26–29, 2005). Ed. by Jean-Daniel Zucker and Lorenza Saitta. Vol. 3607. Lecture Notes in Computer Science. Springer, pp. 165–181. DOI: 10.1007/11527862_12.

Meyer, Bertrand (1992). "Applying "Design by Contract"." *IEEE Computer* 25.10, pp. 40–51. DOI: 10.1109/2.161279.

Mizuki, Takaaki (2016). "Card-based protocols for securely computing the conjunction of multiple variables." *Theoretical Computer Science* 622, pp. 34–44. DOI: 10.1016/j.tcs.2016.01.039.

Mizuki, Takaaki, Isaac Kobina Asiedu, and Hideaki Sone (2013). "Voting with a Logarithmic Number of Cards." In: *12th International Conference on Unconventional Computation and Natural Computation (UCNC 2013)* (Milan, Italy, July 1–3, 2013). Ed. by Giancarlo Mauri, Alberto Dennunzio, Luca Manzoni, and Antonio E. Porreca. Vol. 7956. Lecture Notes in Computer Science. Springer, pp. 162–173. DOI: 10.1007/978-3-642-39074-6_16.

Mizuki, Takaaki and Hiroki Shizuya (2014). "A formalization of card-based cryptographic protocols via abstract machine." *International Journal of Information Security* 13.1, pp. 15–23. DOI: 10.1007/s10207-013-0219-4.

– (2017). "Computational Model of Card-Based Cryptographic Protocols and Its Applications." *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences* 100-A.1, pp. 3–11. DOI: 10.1587/transfun.E100.A.3.

Mizuki, Takaaki and Hideaki Sone (2009). "Six-Card Secure AND and Four-Card Secure XOR." In: *Third International Workshop on Frontiers in Algorithmics (FAW 2009)* (Hefei,

China, June 20–23, 2009). Ed. by Xiaotie Deng, John E. Hopcroft, and Jinyun Xue. Vol. 5598. Lecture Notes in Computer Science. Springer, pp. 358–369. DOI: 10.1007/978-3-642-02270-8_36.

Narodytska, Nina, Toby Walsh, and Lirong Xia (2012). "Combining Voting Rules Together." In: *20th European Conference on Artificial Intelligence (ECAI 2012), including Prestigious Applications of Artificial Intelligence (PAIS-2012), System Demonstrations Track* (Montpellier, France, Aug. 27–31, 2012). Ed. by Luc De Raedt, Christian Bessiere, Didier Dubois, Patrick Doherty, Paolo Frasconi, Fredrik Heintz, and Peter J. F. Lucas. Vol. 242. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 612–617. DOI: 10.3233/978-1-61499-098-7-612.

Niemi, Valtteri and Ari Renvall (1998). "Secure Multiparty Computations Without Computers." *Theoretical Computer Science* 191.1-2, pp. 173–183. DOI: 10.1016/S0304-3975(97)00107-2.

– (1999). "Solitaire Zero-knowledge." *Fundamenta Informaticae* 38.1-2, pp. 181–188. DOI: 10.3233/FI-1999-381214.

Nipkow, Tobias, Lawrence C. Paulson, and Markus Wenzel (2002). *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer. DOI: 10.1007/3-540-45949-9.

Pattinson, Dirk and Carsten Schürmann (2015). "Vote Counting as Mathematical Proof." In: *28th Australasian Joint Conference on Advances in Artificial Intelligence (AI 2015)* (Canberra, ACT, Australia, Nov. 30–Dec. 4, 2015). Ed. by Bernhard Pfahringer and Jochen Renz. Vol. 9457. Lecture Notes in Computer Science. Springer, pp. 464–475. DOI: 10.1007/978-3-319-26350-2_41.

Paulson, Lawrence C. (1989). "The Foundation of a Generic Theorem Prover." *Journal of Automated Reasoning* 5.3, pp. 363–397. DOI: 10.1007/BF00248324.

Pukelsheim, Friedrich (2017). *Proportional Representation – Apportionment Methods and Their Applications*. Springer. DOI: 10.1007/978-3-319-64707-4.

Rastogi, Asem, Nikhil Swamy, and Michael Hicks (2019). "Wys*: A DSL for Verified Secure Multi-party Computations." In: *8th International Conference on Principles of Security and Trust (POST 2019), held as Part of ETAPS 2019: the European Joint Conferences on Theory and Practice of Software* (Prague, Czech Republic, Apr. 6–11, 2019). Ed. by Flemming Nielson and Dave Sands. Vol. 11426. Lecture Notes in Computer Science. Springer, pp. 99–122. DOI: 10.1007/978-3-030-17138-4_5.

Regenwetter, Michel and Bernard Grofman (1998). "Approval Voting, Borda Winners, and Condorcet Winners: Evidence from Seven Elections." *Management Science* 44.4, pp. 520–533.

Richter, Fabian (2021). "Automated Verification and Generation of Voting Rules Using Composable Modules." Master's Thesis. KASTEL Beckert, Karlsruhe Institute of Technology (KIT).

Rivest, Ronald L. (2008). "On the notion of 'software independence' in voting systems." *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 366 (1881), pp. 3759–3767. DOI: 10.1098/rsta.2008.0149.

Rivest, Ronald L. and Warren D. Smith (2007). "Three Voting Protocols: ThreeBallot, VAV, and Twin." In: *2007 USENIX/ACCURATE Electronic Voting Technology Workshop (EVT'07)* (Boston, MA, USA, Aug. 6, 2007). Ed. by Ray Martinez and David A. Wagner. USENIX Association. URL: https://www.usenix.org/conference/evt-07/three-voting-protocols-threeballot-vav-and-twin.

Ryan, Peter Y. A., David Bismark, James Heather, Steve A. Schneider, and Zhe Xia (2009). "Prêt à voter: a voter-verifiable voting system." *IEEE Transactions on Information Forensics and Security* 4.4, pp. 662–673. DOI: 10.1109/TIFS.2009.2033233.

Ryan, Peter Y. A., Peter B. Rønne, and Vincenzo Iovino (2016). "Selene: Voting with Transparent Verifiability and Coercion-Mitigation." In: *Financial Cryptography and Data Security - FC 2016 International Workshops, BITCOIN, VOTING, and WAHC, Revised Selected Papers* (Christ Church, Barbados, Feb. 26, 2016). Ed. by Jeremy Clark, Sarah Meiklejohn, Peter Y. A. Ryan, Dan S. Wallach, Michael Brenner, and Kurt Rohloff. Vol. 9604. Lecture Notes in Computer Science. Springer, pp. 176–192. DOI: 10.1007/978-3-662-53357-4_12.

Sarwate, Anand D., Stephen Checkoway, and Hovav Shacham (2013). "Risk-Limiting Audits and the Margin of Victory in Nonplurality Elections." *Statistics, Politics, and Policy* 4.1, pp. 29–64. DOI: 10.1515/spp-2012-0003.

Satterthwaite, Mark Allen (1975). "Strategy-proofness and Arrow's conditions: Existence and correspondence theorems for voting procedures and social welfare functions." *Journal of Economic Theory* 10.2, pp. 187–217. DOI: 10.1016/0022-0531(75)90050-2.

Scheben, Christoph and Peter H. Schmitt (2011). "Verification of Information Flow Properties of Java Programs without Approximations." In: *International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2011), Revised Selected Papers* (Turin, Italy, Oct. 5–7, 2011). Ed. by Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov. Vol. 7421. Lecture Notes in Computer Science. Springer, pp. 232–249. DOI: 10.1007/978-3-642-31762-0_15.

Shankar, Natarajan (2009). "Automated Deduction for Verification." *ACM Computing Surveys* 41.4, 20:1–20:56. DOI: 10.1145/1592434.1592437.

Shinagawa, Kazumasa and Takaaki Mizuki (2019). "Secure Computation of Any Boolean Function Based on Any Deck of Cards." In: *13th International Workshop on Frontiers in Algorithmics (FAW 2019)* (Sanya, China, Apr. 29–May 3, 2019). Ed. by Yijia Chen, Xiaotie Deng, and Mei Lu. Vol. 11458. Lecture Notes in Computer Science. Springer, pp. 63–75. DOI: 10.1007/978-3-030-18126-0_6.

Shlyakhter, Ilya (2007). "Generating Effective Symmetry-Breaking Predicates for Search Problems." *Discrete Applied Mathematics* 155.12, pp. 1539–1548. DOI: 10.1016/j.dam.2005.10.018.

Smith, Adam M., Eric Butler, and Zoran Popovic (2013). "Quantifying over Play: Constraining Undesirable Solutions in Puzzle Design." In: *International Conference on the Foundations of Digital Games (FDG 2013)* (Chania, Crete, Greece, May 14–17, 2013). Ed. by Georgios N. Yannakakis, Espen Aarseth, Kristine Jørgensen, and James C. Lester. Society for the Advancement of the Science of Digital Games, pp. 221–228. URL: http://www.fdg2013.org/program/papers/paper29_smith_etal.pdf.

Smith, Warren D. (2004). *Cryptography meets voting*. Tech. rep. The Pennsylvania State University. URL: https://www.rangevoting.org/WarrenSmithPages/homepage/cryptovot.pdf.

Sompolinsky, Yonatan, Shai Wyborski, and Aviv Zohar (2018). "PHANTOM and GHOSTDAG: A Scalable Generalization of Nakamoto Consensus." *IACR Cryptology ePrint Archive*. Report 2018/104.

Stark, Philip B. (2010). "Super-Simple Simultaneous Single-Ballot Risk-Limiting Audits." In: *Workshop on Electronic Voting Technology / Workshop on Trustworthy Elections (EVT/WOTE '10)* (Washington, D.C., USA, Aug. 9–10, 2010). Ed. by Douglas W. Jones, Jean-Jacques Quisquater, and Eric Rescorla. USENIX Association, pp. 1–16. URL: https://www.usenix.org/conference/evtwote-10/super-simple-simultaneous-single-ballot-risk-limiting-audits.

Stark, Philip B. and Vanessa Teague (2014). "Verifiable European Elections: Risk-limiting Audits for D'Hondt and Its Relatives." *USENIX Journal of Election Technology and Systems (JETS)* 1.3, pp. 18–39. URL: https://www.usenix.org/jets/issues/0301/stark.

Steinriede, Marion Rabea (2021). "Distance Rationalization for Modular Construction of Verified Voting Rules." Bachelor's Thesis. KASTEL Beckert, Karlsruhe Institute of Technology (KIT).

Swamy, Nikhil, Catalin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin (2016). "Dependent types and multi-monadic effects in F." In: *43rd Annual SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)* (St. Petersburg, FL, USA, Jan. 20–22, 2016). Ed. by Rastislav Bodík and Rupak Majumdar. Association for Computing Machinery, pp. 256–270. DOI: 10.1145/2837614.2837655.

Taagepera, Rein (2002). "Designing Electoral Rules and Waiting for an Electoral System to Evolve." In: *The Architecture of Democracy: Constitutional Design, Conflict Management, and Democracy*. Ed. by Andrew Reynolds. .II: Presidentialism, Federalism and Decentralization, and Electoral Systems. Oxford University Press. Chap. 10. DOI: 10.1093/0199246467.003.0010.

Tang, Pingzhong and Fangzhen Lin (2009). "Computer-aided proofs of Arrow's and other impossibility theorems." *Artificial Intelligence* 173.11, pp. 1041–1053. DOI: 10.1016/j.artint.2009.02.005.

Truderung, Tomasz (Feb. 28, 2019). *Polyas 3.0 E-Voting System Variant for the GI 2019 Election*. Tech. rep. Version 0.9. Polyas GmbH.

– (July 2, 2021). *Polyas 3.0 Verifiable E-Voting System*. Tech. rep. Version 1.2.0. Polyas GmbH.

Verity, Florrie and Dirk Pattinson (2017). "Formally Verified Invariants of Vote Counting Schemes." In: *Australasian Computer Science Week Multiconference (ACSW 2017)* (Geelong, Australia, Jan. 31–Feb. 3, 2017). Association for Computing Machinery, 31:1–31:10. DOI: 10.1145/3014812.3014845.

Vorobyov, Kostyantyn and Padmanabhan Krishnan (2012). "Combining Static Analysis and Constraint Solving for Automatic Test Case Generation." In: *Fifth International Conference on Software Testing, Verification and Validation (ICST 2012)* (Montreal, QC, Canada, Apr. 17–21, 2012). Ed. by Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche. IEEE Computer Society, pp. 915–920. DOI: 10.1109/ICST.2012.196.

Xia, Lirong (2013). "Designing Social Choice Mechanisms Using Machine Learning." In: *International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS '13)* (Saint Paul, MN, USA, May 6–10, 2013). Ed. by Maria L. Gini, Onn Shehory, Takayuki Ito, and Catholijn M. Jonker. International Foundation for Autonomous Agents and Multiagent Systems (IFAAMAS), pp. 471–474. URL: https://dl.acm.org/doi/10.5555/2484920.2484995.

Young, Hobart Peyton (1974). "An Axiomatization of Borda's Rule." *Journal of Economic Theory* 9.1, pp. 43–52. DOI: 10.1016/0022-0531(74)90073-8.

Appendix

A

Card Protocols and KWH Trees

This appendix prints the protocols and KWH trees that the formal method in Chapter 4 found or that we devised based on the evidence from its findings, namely in Algorithm A.1 a standard-deck four-card protocol with its KWH tree in Figure A.1, and a shorter version of a two-color deck five-card protocol in Figure A.2.

Algorithm A.1 Our four-card AND protocol. The first bit is in basis $\{1,2\}$, the second in $\{3,4\}$, and the output in $\{1,2,3,4\} \setminus \{v_2, v_3\}$, where v_2, v_3 are the last two revealed symbols. See Figure A.1 for a KWH tree representation.

1 (shuffle, $\langle(1\ 2\ 3\ 4)\rangle$)
2 v_1 := (turn, $\{1\}$)
3 **if** $v_1 = 1$ **then** (perm, $(3\ 4)$)
4 **else if** $v_1 = 2$ **then** (perm, $(2\ 3\ 4)$)
5 **else if** $v_1 = 3$ **then** (perm, $(2\ 4\ 3)$)
6 **else if** $v_1 = 4$ **then** (perm, $(2\ 3)$)
7 Let $\pi := (1\ 3)(2\ 4)$
8 **repeat**
9 (shuffle, $\langle(1\ 2\ 3\ 4)\rangle$)
10 v_2 := (turn, $\{1\}$)
11 **until** $v_2 = \pi(v_1)$
12 (shuffle, $\langle(2\ 3\ 4)\rangle$)
13 v_3 := (turn, $\{2\}$)
14 Let $\sigma := (1\ 4)(2\ 3)$
15 **if** $v_3 = \sigma(v_2)$ **then** (result, $4, 3$)
16 **else** (result, $3, 4$)

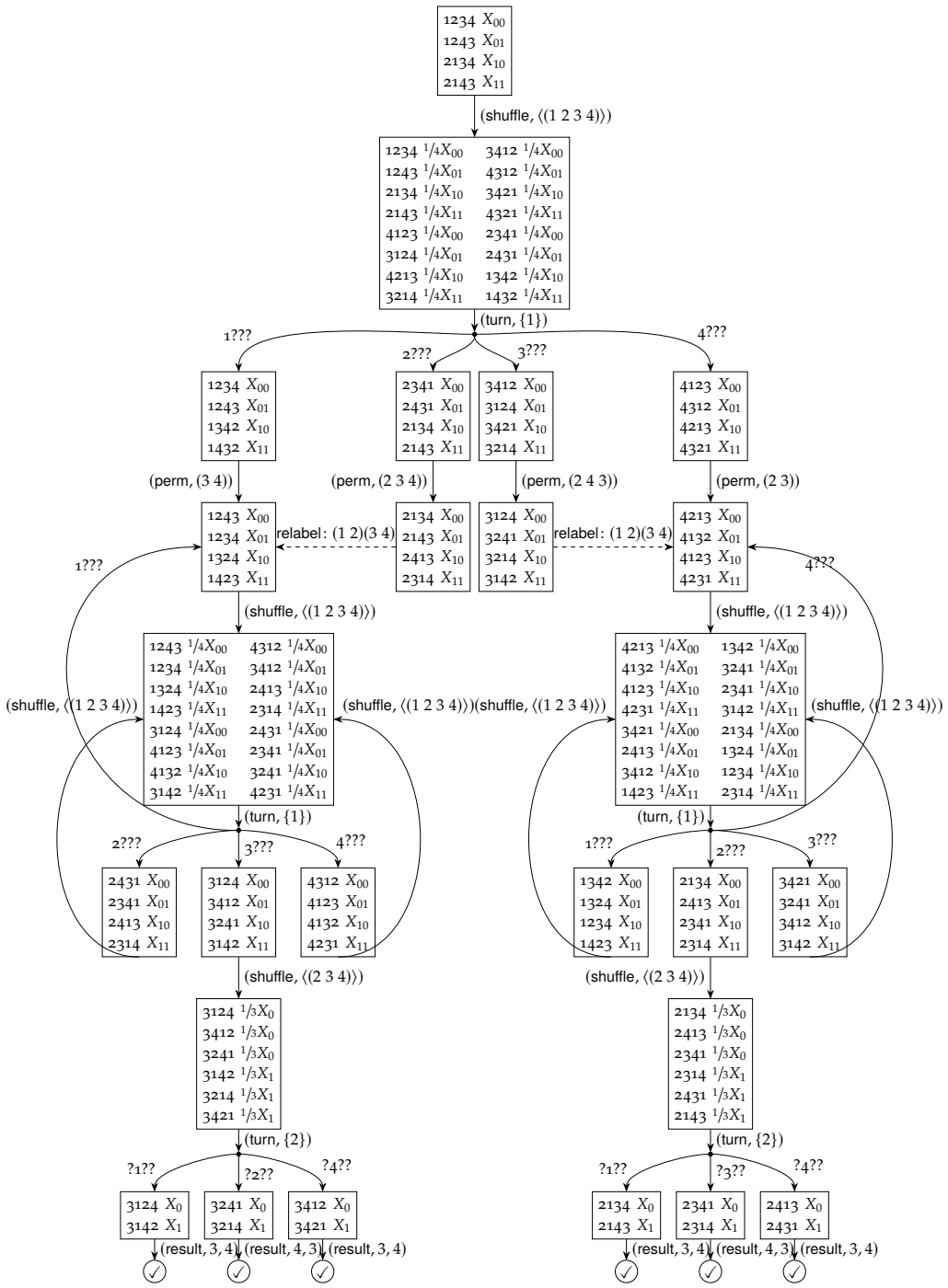


Figure A.1: Four-card Las Vegas AND protocol using random cuts, cf. Algorithm A.1. Here, $X_0 := X_{00} + X_{01} + X_{10}$ and $X_1 := X_{11}$. The relabeling operations are not actual actions to be performed but help abbreviate the write-up of the protocol.

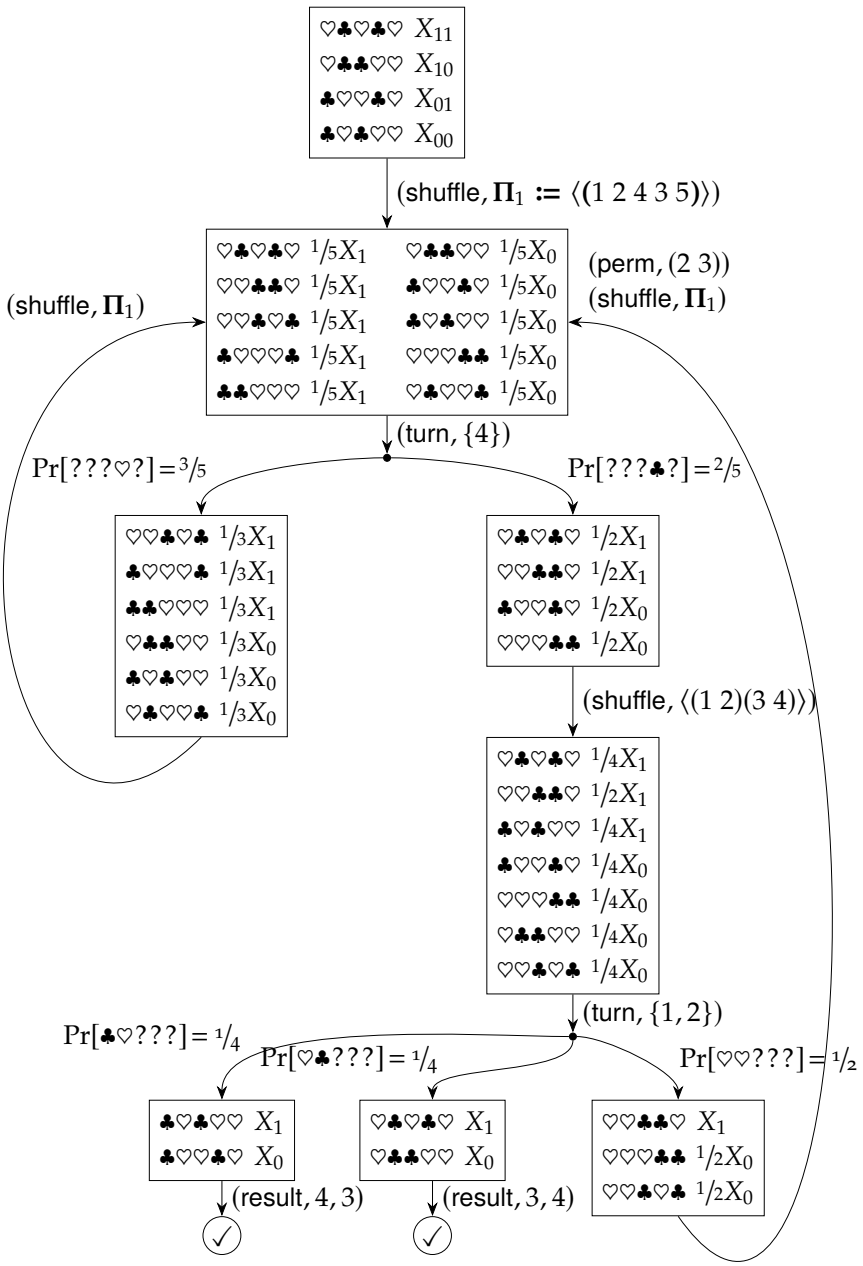


Figure A.2: A shorter version of the five-card two-color Las Vegas AND protocol with uniform closed shuffles given by Abe et al. (2018), which we found when trying to prove the run-minimality w. r. t. closed five-card two-color AND protocols. Here, we save one initial permutation step at the cost of using the slightly more complex shuffle Π_1 that is not as easy to perform as just cutting the cards (albeit still a *random cut*, i.e., a cyclic group generated by a cycle). As we assumed single-card turns, the two-card turn step in the end can be split into two single-card turns, where turning the first card can already result in the final state on the left, and hence the shortest run has only four steps.

B

Rule Construction Graph

This appendix consists of the Isabelle session graph for the framework for the verified construction of voting rules described in Chapter 5 that we then instrument for automatic synthesis. The graph sticks to the names for voting rules that are well-known from literature and denotes the composition structures in a (mostly) self-explanatory manner. The upper three nodes of the graph are the logical core in the Isabelle/HOL system.

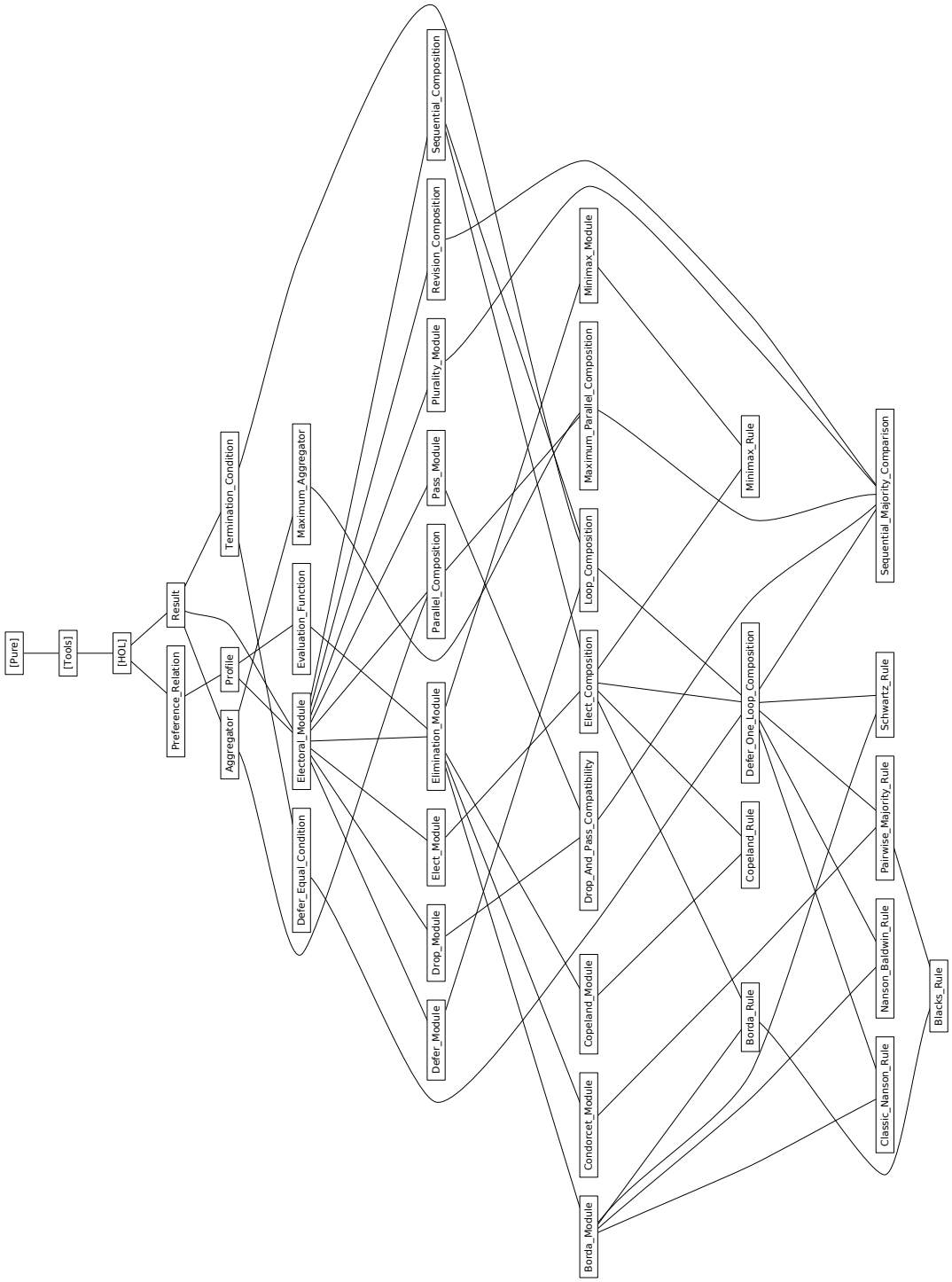


Figure B.1: The Isabelle session graph for the construction framework.

Index