The E2E-VIV Report

Many People

July 7, 2015

Contents

Acknowledgments

This project and report would not have been possible without the commitment and tireless hard work of the team at Galois, Inc. Our special acknowledgment and appreciation goes most especially to Joseph Kiniry, who brought his decades of knowledge, skill, experience and leadership to the project, broadened its scope and led the technical team and writing; and with him, the Galois team members Daniel Zimmerman, Daniel Wagner, Philip Robinson, Adam Foltzer, Shpatar Morina and Leah Daniels. We are indebted to Rob Wiltbank for the Galois engineering contribution, and the long leash he gave to this project.

We would also like to thank the research and technical members of the E2E-VIV Project Team for their contributions to this project from its conception to its completion, with special thanks to Josh Benaloh, Candice Hoke, Keith Instone, David Jefferson, Doug Jones, Aggelos Kiayias, Judith Murray, Ron Rivest, Barbara Simons, and Poorvi Vora.

Equally vital and integral to this report were the reflections, insights and advice from election officials who joined our team, most especially Lori Augino, Judd Choate, Dana Debeauvoir, Mark Earley, Stuart Holmes, Dean Logan, Tammy Patrick, Roman Montoya, and Lois Neuman.

We also thank Sean Beggs, Randall Trzeciak, and Andrew Wasser, Carnegie Mellon University, Heinz College Master of Information Systems Management for their support through the CMU ISM Capstone Program.

We are grateful for the generous financial support of The Democracy Fund, as well as their support of collaborative efforts in the realm of civic tech development.

- Susan Dzieduszycka-Suinat, President and CEO, U.S. Vote Foundation
July 4, 2015

For additional information on U.S. Vote Foundation, please visit www.usvotefoundation.org.

For additional information on the Overseas Vote Initiative, please visit www.overseasvote.org.

For additional information on Galois, Inc., please visit www.galois.com.

For additional information on the Democracy Fund, please visit www.democracyfund.org.

Chapter 1

Introduction

Societies have conducted elections for thousands of years, but technologies used to cast and tally votes have varied and evolved tremendously over that time. In 2015 much of our communication takes place online, and many people want elections to follow this trend. Overseas voters are particularly interested in an online approach, as their voting process often requires extraordinary effort.

In March 2013, Overseas Vote Foundation's President and CEO began a discussion with a small group of election integrity advocates to explore what they would do if faced with the challenge of defining an Internet voting system. Despite their concerns about the security of Internet voting efforts to date, they agreed that addressing this challenge is extremely important.

Obstacles to Internet voting range from the risk of hacking to political and policy considerations. Scientists, federal agencies, cybersecurity specialists, and certain organized activists in the U.S. have urged against exposing the ballots of the most powerful nation on Earth to the seemingly endless range of threats that lurk on today's Internet. Election integrity advocates say that secure, tested, certified remote voting systems are not available. Cybersecurity specialists do not consider online ballot return systems secure, and no Internet voting systems have been certified at the time of this writing. Many election administrators have turned to email as a method for ballot transmission, although it does not provide the benefits that a secure, full-featured voting system would. Email is not secure, yet election administrators and voters regularly use it to transmit ballots because no viable alternatives are available.

Existing vendors of Internet voting technologies, whose systems are neither tested nor certified, would like to openly market and sell their systems within the U.S. without meeting resistance from election integrity advocates. A lack of agreement on how to proceed, a long history of mediocre attempts, ongoing animosity among stakeholders, and a general lack of research on the current questions have soured many people on the hope of using the Internet for voting.

Even so, election officials often want to consider Internet-based technologies so they can serve their constituents in modern and efficient ways. In the current economic climate, investment in election innovation is rare. Election officials in the U.S. are stuck with old technology. They devote scarce resources to support outdated voting systems, and they lack the means to certify new ones that would modernize the voting process.

In order to serve all voters, including overseas citizens, military members, and people with disabilities, new and better ways to use technology for voting are needed.

A Better Technology

In 1981, security researcher David Chaum published an influential paper describing several applications of *public-key encryption*, a new technology at that time. He suggested a way to use public-key encryption to make a set of ballots anonymous while allowing any observer to verify the accuracy of the tally. This was the first step in the development of *end-to-end verifiable* (E2E-V) election technologies.

Over the following decades, numerous researchers published papers describing and refining election systems that use E2E-V technologies. A unique trait of E2E-V election systems is that they allow voters to verify that their own votes have been accurately recorded. At the same time, they allow any observer to verify that all recorded votes have been accurately counted.

Developers designed many E2E-V election systems for a variety of settings—in-person and remote voting, paper-based and electronic voting, simple majority and instant run-off, etc. Early designs were cumbersome, while more recent E2E-V election systems are easier to use.

Election professionals appreciate the benefits that end-to-end verifiability can bring to election systems. Some are taking early steps toward large-scale deployment of E2E-V election technologies for in-person voting systems. However, it is not yet clear whether the benefits of E2E-V technologies can adequately address the legitimate security concerns inherent to Internet voting.

A Proposed Study and Objectives: The E2E-VIV Project

Overseas Vote Foundation (OVF),[1] the leading nonpartisan, nonprofit organization dedicated to overseas and military voter participation, developed and wrote the proposal for this project. On December 19, 2013, it was launched as the End-to-End Verifiable Internet Voting: Specification and Feasibility Assessment Study (E2E-VIV Project). The Democracy Fund, a Washington D.C.-based philanthropic organization whose stated objective is to "...invest in organizations working to ensure that our political system is responsive to the priorities of the American public and has the capacity to meet the greatest challenges facing our country", funded the project.

The proposed project had these goals:

- Convene a diverse group of stakeholders to constructively address open questions about how to vote securely online;
- Define a "whole-product" specification for a secure End-to-End Verifiable Internet Voting (E2E-VIV) system;
- Define a set of testing specifications to demonstrate E2E-VIV security;
- Provide a set of guidelines for system usability, accessibility, and testing;
- Produce a report that examines the feasibility of creating a secure E2E-VIV system;
- Include consideration of legal and administrative challenges, and ballot secrecy, privacy, and confidentiality; and
- Release all information produced to the public.

In addition, an unofficial objective of the project was to "change the conversation" about Internet voting by starting a new, constructive dialogue among computer scientists, usability experts, election integrity advocates, and local election officials from key counties around the U.S.

For the purposes of the E2E-VIV project, *end-to-end verifiable* is defined as follows. First, every voter can check that his or her ballot is cast and recorded as he or she intended. Second, anyone can check that the system has accurately tallied all of the recorded ballots.

[1] Overseas Vote Foundation has since been renamed as U.S. Vote Foundation, and their overseas voter program is now referred to as Overseas Vote, an initiative of U.S. Vote Foundation. For the purposes of this project and report, however, we continue to refer to the organization as Overseas Vote Foundation (OVF).

Some concerns about Internet voting are justified. Internet voting takes all the problems with current remote voting systems and adds to them all the security vulnerabilities of the Internet. No participant in this project discounts these concerns or views E2E-V as a magic fix that makes the Internet secure. We believe that E2E-V properties, which apply even when voters use potentially untrusted devices like personal computers and transmit votes over an untrusted medium like the Internet, are important for Internet voting. The E2E-VIV Project does not attempt to make the Internet secure. Instead, it examines whether E2E-V can effectively counter the risks of Internet voting while bringing substantial new benefits not found in today's voting systems.

The E2E-VIV Project also aims to clear up a misunderstanding in the U.S. election community. Our country's scientists are not against technology advancements, nor are they inherently at odds with election officials who seek technology improvements to meet their administrative challenges. Instead, scientists doubt claims of security regarding existing systems that are not publicly tested or vetted. The scientific leaders on this project have often pointed out security vulnerabilities in past systems; however, the E2E-VIV Project has led them to agree that if Internet voting can happen, it must be in a system that takes advantage of end-to-end verifiability and auditability.

Shared Goals

Election officials and scientists involved in elections share the same overall goals: that voting systems provide accurate results, protect voters' privacy, are easy to use for all voters, and are robust against both accidental and intentional disruptions. Additionally, election officials must be able to show the public that their voting systems achieve these goals.

This project shows that many scientists care deeply about addressing the needs of election officials as they serve remote voters. These scientists are highly motivated to help when given an opportunity, and they would like to work with election administrators to examine the possibilities in this realm. This project also shows that scientists could improve their understanding of the practical issues that election officials face and find better ways to collaborate and communicate with them to develop technical solutions.

Success

We hope, that in addition to the concrete outcomes of this project, our research and testing-based approach to examining Internet voting will move beyond the current stalemate and stimulate election development overall. The election industry continues to operate in a traditional way, with only a few vendors able to survive despite demand to move away from outdated, expensive, hardware-oriented solutions. A successful outcome of this project would:

- Help build a specification for a usable, secure E2E-VIV technology;
- Determine the strengths and weaknesses of such a system; and
- Identify reasons to pursue or not pursue this approach.

Ideally, the resulting specification would be:

- Supported by the vast majority of the contributors, including the technical, usability, testing, and research teams;
- Endorsed by the vast majority of the E2E-VIV Project advisory council; and
- Endorsed by the major stakeholders in elections administration as represented by the project's local election officials.

It was acknowledged from the outset that if the project team determined that current techniques were weak and should not be pursued, this would also be an outcome with many useful implications.

The E2E-VIV Project hopes to receive support and endorsement from many members of the election integrity community, as represented by key members of the Election Verification Network, the Verified Voting Foundation and beyond. We intend for the specification to fulfill the following requirements:

Independent Implementation The specification must have sufficient detail and clarity that an independent party can implement the election system without having extensive dialogue with project participants.

Independent Validation It must be possible for a moderately proficient IT expert to objectively determine, in a reasonable time frame and at reasonable cost, whether a constructed election system fulfills the specification.

Evidence-Based Decisions Every decision made in the crafting of the specification must be objectively justifiable and the evidence for the decision must be traceable.

Scope

The original project was limited to system specification and testing only. The project participants did not envision system development in Phase I beyond mock-ups to help test usability. That changed, however, when OVF engaged Galois, Inc. to conduct the technical project management. Galois's management offered to donate engineering time to the project to build "demonstrators" that would be used to prove the concepts of E2E-V and to further examine security and usability. This additional work broadened the scope of the project.[2]

The demonstrators developed using Galois research and development funding are:

- Developed in a completely transparent and public manner within the Galois GitHub Organization;
- Cross-referenced, and thus traceable to and from, all specification aspects (from domain models to behavioral design specifications);
- Replicated into the E2E-VIV GitHub Organization;
- Licensed under either a mainstream Open Source license with a strong community or an alternative license tuned to the elections community.

1.1 Project Team

The E2E-VIV Project provided an opportunity to combine the abilities, knowledge, experience, and expertise of a diverse group of technologists, computer scientists, and election officials involved in election integrity. Technical, usability, testing, and local election official sub-teams were created to make communication easier. The project also established an advisory council to open communication with interested members of the election community.

OVF, as the official grantee, was responsible for the overall project conception, proposal development, presentations, communications, management, team recruitment, contractual obligations, public relations, events, and budgeting.

Galois, Inc. provided the technical and engineering project management, wrote much of and edited the final report, and facilitated the communication and decision-making of the team.[3]

[2] The Galois engineers developed a set of rigorous engineering demonstrators that can be refined to fit into a working election system. Third parties can perform independent verification and validation of these demonstrators.

[3] See the *GaloisInc/e2eviv* GitHub repository at `https://github.com/GaloisInc/e2eviv` to best understand exactly what Galois's contributions are.

1.1.1 Team Members

Layout and editorial note: We will be turning this section into a more attractive illustration to enumerate participants in the project. It will denote exactly who participated and in what fashion. No team member will be listed without their consent.

Project Manager: Susan Dzieduszycka-Suinat, Overseas Vote Foundation

Lead Technical Project Manager: Dr. Joseph Kiniry, Galois

Technical Team

Dr. Josh Benaloh Senior Cryptographer, Microsoft Research

Dr. David R. Jefferson Lawrence Livermore National Laboratory

Dr. Doug W. Jones Associate Professor, Department of Computer Science, University of Iowa

Dr. Aggelos Kiayias Associate Professor, Computer Science and Engineering, University of Connecticut

Dr. Olivier Pereira Professor, Institute of Information and Communication Technologies, Electronics and Applied Mathematics, Ecole Polytechnique de Louvain

Dr. Poorvi Vora Associate Professor, Department of Computer Science, The George Washington University

Dr. David Wagner Professor, EECS Computer Science Division, University of California Berkeley

Dr. Dan Wallach Professor, Department of Computer Science, Rice University

Usability Team

- Keith Instone, User Experience Consultant
- Morgan Miller, Usability Analyst, Experience Lab
- Dr. Judith Murray, Research Consultant

Election Auditing

Dr. Philip Stark Professor and Chair of Statistics, University of California Berkeley

Testing Team

Dr. Duncan Buell Professor of Computer Science and Engineering, University of South Carolina

Andrew Regenscheid Mathematician, National Institute of Standards and Technology

Advisory Council

Dr. Ben Adida

Dr. Michael Clarkson Assistant Professor of Computer Science, The George Washington University

Dr. J. Alex Halderman Assistant Professor of Computer Science and Engineering, University of Michigan

Candice Hoke Professor of Law, Cleveland State University

Dr. Ron Rivest Vannevar Bush Professor of Computer Science, Massachusetts Institute of Technology

Noel Runyan Primary Consultant, Personal Data Systems

Dr. Peter Ryan Professor in Applied Security, University of Luxembourg

Dr. Barbara Simons Research Staff Member, IBM Research (retired)

Dr. Vanessa Teague Research Fellow, Department of Computing and Information Systems, University of Melbourne

John Wack Voting Systems Standards, National Institute of Standards and Technology

Dr. Filip Zagorski Assistant Professor of Computer Science, Wroclaw University of Technology

Local Election Officials

Lori Augina Director of Elections, Washington State, Secretary of State

Rachel Bohman Former Hennepin County Elections Manager (Minnesota)

Judd Choate Director of Elections, Colorado, Secretary of State

Dana Debeauvoir Travis County Clerk (Texas)

Mark Earley Voting Systems Manager, Leon County (Florida)

Dean Logan Los Angeles Registrar-Recorder/County Clerk (California)

Stuart Holmes Election Information Systems Supervisor, Office of the Secretary of State (Washington)

Dr. Lois H. Neuman Chair, Board of Supervisors of Elections, City of Rockville (Maryland)

Roman Montoya Deputy County Clerk, Bernalillo County (New Mexico)

Tammy Patrick Senior Advisor to the Democracy Project, Bipartisan Policy Center and Former Federal Compliance Officer Maricopa County (Arizona)

Overseas Vote Foundation Support Team

Susan Dzieduszycka-Suinat President and CEO

Paul McGuire Legal Counsel and Secretary of the Board

Richard Vogt Treasurer and Chief Financial Officer

Capstone Project Team, Carnegie Mellon University, Heinz College, School of Information Systems & Management; Master of Information Systems Management and Master of Science in Information Security Policy and Management: in early 2014, a Capstone Team was assigned to the project team to assist on the Comparative Analysis of E2E systems.

1.1.2 Stakeholder Groups

Although not on the official project team, several communities relevant to the E2E-VIV Project have offered essential input. These communities include:

Election Verification Advocates Election verification advocates are numerous, well-informed, and strongly connected. They care deeply about election integrity and verifiability. Internet voting is a high-priority issue for many of them.

Election verification advocates tend to be skeptical about Internet voting. This is because several elections vendors have developed Internet voting products that are proprietary, closed-source, and not verifiable. Their products have never had a public audit. Many vendors nonetheless make claims about the security of their products, and election verification advocates reject these claims. Some vendors recommend that elections be outsourced entirely to their own companies—a condition that will never be acceptable to the election verification community, even for an E2E-VIV system.

Some election integrity advocates are in favor of verifiable Internet voting, others are adamantly against Internet voting of any kind, but the bulk of them are undecided. That majority recognizes that significant scientific and engineering challenges must be met to design and develop an Internet voting system. They recognize that the decision to deploy such a system is very much a subjective, political one. In some contexts, such as deciding the winner on a reality TV show, some advocates view using a non-verifiable, outsourced election apparatus as an acceptable choice. However, for government elections of any value, such an option is unacceptable to virtually every advocate.

Being fully transparent with—and listening to the feedback from—the election verification advocate community is mandatory. If the evidence presented in this report does not sway the bulk of that community, the pursuit of any next phase in this project will be contentious.

Standards Bodies Perhaps surprisingly, little national or international election standardization exists. An effort to standardize data interchange formats began about a decade ago and eventually collapsed after agreement on one small standard. This first effort failed for many reasons. Vendors lobby against, and are uninterested in, interoperability. The Election Assistance Commission (EAC) issued Voluntary Voting System Guidelines (VVSG), but these were not geared toward a component-based approach to system design; since devices could not be plugged together, there was little cause for defining interfaces and data file formats. Also, the election research community did not accept that first effort at standardization.

In 2015, the situation changed with the rebirth of the Institute of Electrical and Electronics Engineers (IEEE) 1622 committee and its focus on elections. The IEEE Voting System Standards Committee 1622 (VSSC/1622) creates standards and guidelines around a common data format for election data. The goal is that future election equipment used in U.S. elections and abroad can interoperate more easily. The Voting System Standards Committee intends for their standards and guidelines to be required in future versions of the VVSG.

Many of the top researchers, election advocates, and election officials in the world participate in this committee. Representatives from the major election systems vendors also participate, because they recognize that interoperability will be mandated by future versions of the VVSG.

Standards are critical to any future work on E2E-VIV systems for several reasons. First, given the compositional nature of most E2E-V systems' designs, it is likely that different subsystems will be created and supported by different organizations. Second, in order to enable arbitrary third party verification, clearly defined common data formats must be used. Third, at some point in the future, if E2E-VIV systems are accepted in the mainstream, the EAC must certify them. The EAC, National Institute of Standards and Technology (NIST), and IEEE standards must recognize their core capabilities, subsystems, interfaces, and what constitutes legitimate evidence of correctness and security. A standard means to document these aspects and evidence is necessary to establish a sound, accurate, and expedited certification process.

Vendors Two types of vendor are relevant to this project: (1) existing vendors of proprietary election systems, and (2) future vendors that support open source election systems.

Existing vendors may be interested in the results of this project, particularly if it moves to a second phase that focuses on the design and development of an open source system. Those that have an existing non-E2E-V Internet voting product (e.g., Scytl, Dominion, and Everyone Counts) could benefit from this work if they take on the tough challenges of verifiability and usability and tackle them accordingly. Any increase in attention on—or any hint of support from the verifiable elections activist community about—Internet voting aligns with their marketing goals.

Vendors can use the requirements contained in this project in various ways. Ideally, they will actively and positively absorb the recommendations. Any investment towards making their commercial systems end-to-end secure and verifiable, as long as there is publicly available evidence of such improvements, would be welcomed by the bulk of the research and activist communities. On the other hand, it is also possible that vendors will simply use these requirements as a checklist, claiming that their systems are E2E-V but providing no evidence. Only time will tell which path each existing vendor chooses to take.

The other type of vendor that is relevant to this project, supporting open source election systems, does not yet exist. We expect such commercial support organizations to emerge to support any E2E-VIV system that occurs as a result of this project. These organizations would not need to have expertise in the underlying cryptography, formal methods, or usability and accessibility. They would rely on the high-assurance development method described in this report, which produces validation and verification artifacts that generate evidence of their correctness and security properties. Additionally, modern formal system specification languages support traceability from requirements to evidence.

Any future mission-critical E2E-VIV system will be robust to any integration, customization, or evolution work by these vendors. That is, systems developed according to the requirements and recommendations in this report cannot be accidentally or purposefully broken without third parties detecting a certification failure. Only these complete, consistent, traceable, evidence-generating artifacts make it possible for such new support vendors to emerge.

Hackers and Hacktivists Hackers and election hacktivists are an important audience for this project. Hackers, constructively or destructively, help make open source systems more secure. Hacktivists catalyze movements of like-minded technical individuals. Across the world, their attention has brought to light the flaws of insecure and incorrect electronic voting systems.

Within these sub-communities, it is an undisputed precept that the design and development of a secure system used for public good must be open to critique and improvements from the public. Leveraging that attention is especially valuable for nationally critical systems like public elections. Consequently, direct engagement with these groups, while potentially tempestuous at times, is wise and will have many benefits.

Election Officials Local elections officials (LEOs) are the core stakeholders in the design and deployment of E2E-VIV technologies. There are over 10,000 election jurisdictions in the United States. That means there are over 10,000 different ways that elections can be run. Smoothly integrating with existing election processes is important. Basic issues like common data formats are simple technical challenges that have straightforward, though potentially politically delicate, solutions.

The enormous variety of technologies, large and small, at the federal, state, and local levels make deploying any new election technology difficult. Traditional IT practices for the design, development, and maintenance of election software cannot work in this setting. No vendor can support 10,000 variants of a single code base to satisfy 10,000 clients, each with slightly different requirements. Some of the software engineering recommendations in this report, particularly those that touch upon feature modeling and software product lines, are directly relevant to the elections setting.

More subtle challenges, such as ensuring that election officials who are uncomfortable with technology can deploy and support a E2E-VIV system for their overseas, military or disabled voters, have little to do with technical solutions. These problems and solutions have more social, political, and psychological roots, and thus require soft solutions.

Tools such as documentation and tutorials, webcasts and screencasts, reports, and demonstration software have a limited reach. Open Source projects often have friendly online forums to welcome and guide newcomers, as well as regularly scheduled and community-organized developer and user conferences. We hope that kind of open, active and supportive community will evolve around E2E-VIV.

Voters Despite the importance of all of the aforementioned stakeholders, the most important stakeholders—the ones that hold veto power over the wide-scale deployment of E2E-VIV technologies—are the voters. Many voters want flexible, comfortable voting that is not tied to a particular polling place. However, they may not realize the implications of this desire, particularly with regard to security and privacy. Educating voters about the challenges of Internet voting is important, but it is not possible to make every voter aware of the usefulness of, and critical need for, E2E-V election systems.

Based upon early usability studies of E2E-V election systems, we can expect only a small fraction of voters to understand and use the verifiability features of E2E-VIV systems, even if the user experience of these systems is well-designed. Voters' trust in their election apparatus, and how their voting experience affects their trust in their government, is paramount. Consequently, changes in elections that may impact trust are difficult to make.

Voters are sensitive to changes that are bluntly visible, like electronic pollbooks and electronic voting machines. The past decade, with its introduction of electronic voting machines and the subsequent outcry for a return to a "paper trail", has taught some voters not to trust election technology. A vocal minority may misunderstand changes and be hostile or alarmed. Public reactions are unpredictable, as evidenced by the introduction of voter ID laws across the U.S. and the alarm raised by the use of email for ballot return (even in extraordinary circumstances).

Finally, and perhaps most critically, there is a delicate balance between comprehensibility and security—the twin challenges of digital election systems. People view paper-based elections as transparent and comprehensible. Digital elections have lost those properties. However, in the right circumstances, digital elections benefit from tabulation efficiency, decreases in residual vote count, and facilitation of independent voting for voters with disabilities. Internet voting shifts the balance towards voter convenience, but with a significant loss in general comprehensibility; to the first significant digit, the percentage of voters that are cryptographers is zero.

Choosing to design and deploy an election system that is end-to-end secure and based upon cryptographic principles is a policy decision that firmly delegates trust to a precious few: on the front lines, the elected officials and the politicians voting for modernization, and behind the scenes, the cryptographers, scientists, and engineers responsible for the system's design, development, validation, and verification. Those few assume an enormous responsibility.

1.2 Methodology

The technical project management of this first phase of the E2E-VIV project was organized in a workflow that focused on delivering an evidence-based report and its complementary Open Source technical artifacts. The methodology is summarized as follows:

- Absorb all input from team members.
- Read all literature on Internet voting.
- Write baseline business and technical requirements and solicit feedback from technical team.
- Write personas as a foundation for user experience studies.
- Interview local election officials based upon requirements, personas, and their current elections framing.
- Outline report and solicit text and reflections from team members.
- Reflect upon the latest advances in cryptography for E2E-VIV.
- Reflect upon the latest advances for reasoning about cryptographic algorithms, protocols, and implementations.
- Craft a parameterized architecture space that reflects underlying requirements and standard cryptographic protocols.
- Integrate all team member text and reflections and develop a final report table of contents.
- Solicit more text and reflections from team members based upon the final report table of contents.
- Write the remaining unwritten chapters.
- Solicit input from all team members on Part 1.
- Solicit input from technical team members on Part 2.
- Solicit initial reflections from technical team members on the feasibility of Part 2 potential recommendations.
- Gather all input from team members and capture it in appendices, citations, and footnotes.
- Identify useful illustrations and figures for report and communicate them to the illustrator.

- Copy-edit, lay out, polish, and release the report.
- Clean up and make public the GitHub repository that includes the report and all associated artifacts of the project.

The outcomes of this process are this report and the reflections of a body of experts in the domain of verifiable elections, as summarized in the following section.

1.3 Outcomes

This project has produced a high-level system specification in the form of a set of business and technical requirements. Accompanying that set of requirements are recommendations about the underlying means by which those requirements should be fulfilled.

These recommendations focus on the three dimensions necessary to fulfill the strenuous requirements of any E2E-V system: cryptography, architecture, and engineering. The cryptographic foundation is a formal framework in which to evaluate E2E-V cryptographic protocols. The architecture specification is a formal description of an architecture space, defined by several parameters, in which solutions can be designed, built, and evaluated. Finally, the rigorous engineering necessary to create a high-assurance E2E-VIV product is specified via a recommended software engineering process, methodology, and set of technologies. Used properly, these can fulfill the security and technical requirements while generating the necessary evidence to substantiate that the system is fit-for-purpose.

Fulfilling the usability and security requirements would not be sufficient for a positive assessment by the project team. A full system specification that is usable and secure may, for example, be too expensive to build, too difficult to deploy and manage, or mandate too much expertise from election officials to operate. In the end, social non-functional requirements may trump technical functional requirements.

A positive assessment by the majority of the technical team would mean they agree that the specified election system meets all of the requirements set forth by the charter of the group and that, eventually, an open source E2E-VIV system could be developed, tested, and potentially deployed.

A negative assessment by the majority of the technical team would mean they do not agree that the specified election system meets all of the requirements set forth by the charter of the group, and that designing a usable and secure Internet voting system is still an open scientific, not engineering, challenge.

The final assessment of the team is found in Chapter 9.

1.3.1 User Interface Design

The user interface (UI) of the E2E-VIV election system is a critical factor in its acceptance. The system must be simultaneously usable, accessible, and secure. Consequently, a detailed UI design informed by user experience (UX) and accessibility testing is a mandatory component of a future detailed system specification.

This report's system specification is focused on high-level requirements for any E2E-V election system (Internet-based or not). As such, several requirements focus on UI and UX and stipulate the necessary framing for UI/UX designs and studies.

Usability and accessibility studies and testing are key components of this report, and the outcome of this first study is meant to inform the UI design of any future E2E-VIV systems. As such, most of the effort relating to UI and UX within the project has focused on developing a technical infrastructure and complementary process that allows efficient definition and execution of usability and accessibility studies.

Researchers conducted an initial usability study, gathering qualitative feedback from several dozen voters. The results of that study, whose full report is included in Appendix B, led to three conclusions about voters' attitudes regarding Internet voting.

1. *Voters trust the voting system and their election officials.* This trust is both a blessing and a curse. It is a blessing because it means that election officials and the government are working with voters that have a positive attitude; their trust can only be lost and does not need to be won. But it is also a curse because it means that voters will trust Internet voting systems that have no transparency, end-to-end security, or verifiability. As such, this opens the door for vendors to sell their technology and gather naïve voter feedback as "evidence" that their systems are fit-for-purpose for modern public elections, even if that is not the case.

2. *Voters are not interested in learning about verifiability or verification.* Fortunately, only a small fraction of voters needs to actually verify their votes for most E2E-VIV election schemes to generate sufficient independent evidence that the election outcome is correct. The requirements and recommendations reflect this low level of voter interest in verification and suggest several means by which the threshold for election verification may, nevertheless, always be achieved.

3. *Voters expect the Internet voting experience to be somehow different from a traditional voting experience.* Voters expect modern, rich online experiences akin to those they know from popular commercial websites and smart phone applications. This raised expectation opens the door for novel experimentation to develop a "21st Century" voter experience.

Chapter 2

Remote Voting

2.1 Rationale

Remote voting is becoming increasingly common, necessitated by the growing and diverse needs of voters. It is used to enable overseas citizens and military personnel to participate in elections, reduce access related discrimination domestically, and decrease expensive administrative overhead of polling locations.

In the United States, fewer than 5% of ballots cast in general elections during the 1980s were cast before Election Day. By the 2012 general election, 31% of all ballots were cast early, and 17% were cast by mail. The states of Washington, Oregon, and most recently Colorado have entirely switched over to all-mail voting. For an election system to fully enfranchise the electorate, it must treat remote voting as a first-class capability rather than as a backup system with second-class effectiveness, speed, security, and integrity.

2.1.1 Accessibility

According to the International Center for Disability Information and the National Institute on Disability and Rehabilitation Research, 20% of Americans live with disabilities. The Voting Accessibility for the Elderly and Handicapped Act of 1984 mandates that any person with a disability may vote remotely without having to present medical documentation, reducing the barriers to remote voting for those with accessibility needs.

2.1.2 Overseas and Military Voters

In 1986, Congress enacted the Uniformed and Overseas Citizens Absentee Voting Act (UOCAVA) to address the needs of citizens in the uniformed services, merchant marines, and other overseas civilians. UOCAVA mandates that these overseas and military voters (UOCAVA voters) be able to register and vote remotely in federal elections. It is difficult to calculate an exact number of UOCAVA eligible voters. According to estimates from the United States Elections Project [143], there were 5,127,418 UOCAVA eligible voters for the 2012 U.S. general election and 5,345,814 for the 2014 U.S. general election.

2.1.3 Domestic Absentee

Domestic absentee voters are those who cast their votes by mail because they are unable, or do not want, to be present at polling locations on Election Day; this excludes UOCAVA voters and voters in states that vote exclusively by mail. According to the Election Administration and Voting Survey [55] published by the U.S. Election Assistance Commission, domestic absentee voters cast 16.6% of the ballots in the 2012 U.S. general election and 17.5% of the ballots in the 2014 U.S. general election. As of this writing 27 states allow voters to apply for an absentee ballot without providing a justification, known as "no-excuse absentee voting".

2.1.4 Expectations

In 1952, a study by the American Political Science Association defined ten voting rights necessary for members of the armed services [20]. Although initially defined for military voters, these rights have served as the basis for defining the expectations of all remote voters through UOCAVA and other subsequent legislation. Among these rights are:

1. To vote without registering in person;
2. To vote without paying a poll tax or having to meet unreasonable requirements;
3. To use the Federal Postcard Application[1] both to register and to request a ballot, rather than having to use state-specific paperwork;
4. To receive ballots for primary and general elections in time to vote;
5. To be protected in the free exercise of their voting rights; and
6. To receive essential information needed to vote.

These rights first found their way into law in the Federal Voting Assistance Act of 1955 (FVAA) but, due to partisan struggles, the rights were watered down from requirements into recommendations; this left most of the decisions about final implementation to the states. Over time, subsequent legislation has strengthened these recommendations into guarantees and requirements and has expanded rights to include participation by non-English speakers and people with disabilities.

2.2 History

2.2.1 Armed Forces Voting

Before the American Civil War, U.S. citizens primarily voted in their places of residence and many states legally barred the casting of votes from outside state borders. There was little effort from any state to accommodate absentee voting. In 1864, however, the Civil War displaced soldiers from their residences and Lincoln's re-election was at risk. With much lobbying on behalf of the Republican Party (and opposition from the Democratic Party), nineteen Union states adopted absentee voting procedures for military voters in time for the election. Since the motivation for passing these laws was to secure Lincoln's re-election rather than permanently expand voting access, many states treated absentee military voter laws as temporary and repealed them after the war.

For the 1918 midterm elections, the U.S. War Department decided that it was not ready to support the military vote; the Department prohibited individual states from canvassing overseas soldiers serving in World War I. World War II inspired another push for the military vote in hopes of supporting the re-election of the presidential incumbent. This prompted the Soldier Voting Act (1942). Although it was passed too late for the 1942 midterm elections, this law gave military personnel absentee voting rights for federal elections during times of war without being required

[1] Federal Postcard Application is the Department of Defense name for the official UOCAVA voter registration and absentee ballot request form.

to pay a voting tax or postage costs. The act has continuing significance: it stated that all overseas voting would be regulated at the federal level and implemented at the state level, a structure that continues to this day. By 1944, partisan politics led to the weakening of the state mandate from a requirement to a recommendation, leading to a 29.1% turnout rate vs. the 60% domestic turnout rate [178].

2.2.2 Remote Civilian Voting

Progress for civilian absentee voters lagged behind progress for military voters. In 1896, states began to introduce civilian absentee voting legislation. By 1924, only three states had no absentee voting legislation. State laws, however, were a confusing and inconsistent patchwork that limited absentee turnout. Major progress for civilian absentee voters would only come with legislation motivated primarily by military voters such as the FVAA.

In the 1960s, lobbying from overseas civilian groups led to amendments to the FVAA. This effort expanded the number of civilians covered by the law, though the amendments were once again voluntary recommendations to the states. As lobbying pressure increased further, the Overseas Citizens Voting Rights Act (OCVRA) passed in 1974. OCVRA was the first law to guarantee, rather than only recommend, absentee voting rights for overseas civilians.

In 1986, legislators passed UOCAVA, which combined and replaced FVAA and OCVRA. UOCAVA made the rights recommended by the previous acts into requirements for both military and overseas civilian voters.

2.2.3 Disabled Civilian Voting

The Voting Rights Act (VRA) of 1965 was the first legislation to allow voters who require assistance to vote—because of blindness, disability, or inability to read or write—to receive help from a person of their choice.

The Voting Accessibility for the Elderly and Handicapped Act of 1984 (VAEHA) improved access for disabled and elderly individuals. Like FVAA and OCVRA, VAEHA did not specify standards of access; individual states set their own standards. VAEHA also limited the disabled voters group to those with *physical disabilities*. It did, however, mandate that "no notarization of medical certification shall be required of a voter with a disability with respect to an absentee ballot or application for such ballot."

The 1990 Americans with Disabilities Act (ADA) requires that people with disabilities have access to basic public services, including the right to vote. ADA does not strictly require that polling locations are accessible, however it did extend the definition of disability to

> "a person who has a physical or mental impairment that substantially limits one or more major life activities, a person who has a history or record of such an impairment, or a person who is perceived by others as having such an impairment."

Over the years, legislators have passed several federal laws to protect voting rights of disabled citizens. The majority of these laws have struggled to clearly define a representative range of disabilities. The laws often focus on in-person access to physical polling locations, which is expensive for states to implement. State-defined policies often ignore rights to voting privacy, and exclude people who have multiple disabilities because technologies that may help them vote are not yet available.

2.2.4 Modern Remote Voting

In 2002, Congress passed the Help America Vote Act (HAVA) in response to problems found in gathering, counting, and auditing ballots in the 2000 presidential election. HAVA requires that all polling places in elections for federal office anywhere in the United States have at least one voting system capable of assisting disabled voters. This requirement addresses some accessibility concerns.

HAVA was also a response to the large number of rejected ballots in the 2000 election and an inability to sufficiently audit ballots. HAVA recommends that election systems produce a Verifiable Voter Paper Audit Trail (VVPAT) while preserving the privacy of the voter and the secrecy of the cast ballot. HAVA also created the United States Election Assistance Commission (EAC) to oversee the development of new voting machine standards. The National Institute of Standards and Technology (NIST) released the Voluntary Voting System Guidelines (VVSG) to aid in this transition.

The Military and Overseas Voter Empowerment Act of 2009 (MOVE) addresses barriers to overseas voter participation, and specifically attempts to reduce the number of ballots that are not counted due to late receipt. MOVE requires states to send absentee ballots at least 45 days before Election Day, make all registration material and blank ballots available electronically, and remove notarization requirements on voting applications and ballots.

Because state governments enforce existing voting regulations, these regulations are hampered by local political attitudes. In 2010, the Uniform Law Commission oversaw drafting of the Uniform Military Services and Overseas Civilian Absentee Voters Act (UMOVA). UMOVA is designed to identify and standardize the important protections and benefits found in federal legislation like UOCAVA and MOVE in state and local elections. As of April 2015, fourteen states and the District of Columbia have enacted UMOVA.

2.3 Shortcomings of Current Practice

Despite years of progressively stronger legislation addressing the needs of remote voters, many shortcomings still exist in current election practices. The topics listed below draw from specific concerns that negatively affect remote voting participants.

2.3.1 Use of Communication Technologies

The majority of remote voting takes place via postal mail, which has many inherent faults that affect the voting process. The 2008 Post-Election UOCAVA Survey Report and Analysis found that voting boards did not count 52% of attempted UOCAVA votes due to problems in the mail delivery process. In addition, maintaining correct voter registration information for military voters and others who frequently change addresses while abroad is expensive and prone to error.

To address differences between states' absentee registration and voting practices, the Federal Voting Assistance Program (FVAP) provides a Voting Assistance Guide (VAG) to support UOCAVA voter registration and ballot request. The VAG supports military Voting Action Officers who assist their units with voter registration instructions.

Since MOVE was passed in 2009, online registration has expanded significantly. OVF and the FVAP offer online voter services to UOCAVA voters through their websites, as do many states and counties. Online blank ballot transmission is becoming routine, and email communications between election officials and voters are flourishing.

Unfortunately, the problem of late ballot receipt and rejection has not been solved.

2.3.2 Accessibility and Usability

In 2007, 20% of Americans with disabilities said they were unable to vote in a presidential or congressional election due to difficulty getting to the polls or barriers at polling locations [170]. The voting technologies used, and the physical locations of polling places, often cause such problems. In the 2000 presidential election, 56% of randomly sampled polling places in the United States had at least one barrier to disabled voters [150].

Voters with disabilities often forfeit privacy in order to have someone help them with in-person voting. Polling locations often lack technology that can help, and the assistive technology that is available is often too difficult for voters and poll workers to use. Remote voting still presents obstacles: those with dexterity impairments often have problems handling and marking paper absentee ballots.

2.3.3 Auditing

Although voter fraud is fairly uncommon, it is a major concern in a bipartisan system. Voter fraud is very difficult to detect without reducing turnout or disenfranchising legitimate voters. Policies intended to reduce fraud or protect identities, such as increasingly prevalent and strict voter ID requirements, often lead to a higher rate of rejected ballots. This is the case even for remote voters. In the 2012 general election, voting boards rejected over 20% of absentee ballots due to non-matching signatures or insufficient identification [55].

2.3.4 Voter Privacy

A fair voting system must ensure voter privacy. Privacy promotes voter independence and helps prevent voter coercion and vote buying. Most remote voting practices require that voters forfeit independence or privacy because election officials cannot enforce privacy when voting takes place outside of polling locations. Several states require voters to sign a voter privacy waiver when casting a remote ballot [179].

Chapter 3

E2E-VIV Explained

3.1 Election Process and Goals

A typical Internet voting election process has six phases:

Setup During the setup phase, election officials gather information needed to run the election. This includes:

- gathering registration information for all voters;
- identifying the issues and races that will be voted on;
- designing ballots, often in multiple languages, for all precincts participating in the election;
- sending instructions and other information about the election to voters; and so on.

Distribution Different voting systems use different mechanisms to distribute ballots to voters, such as postal mail, email, or a website that provides downloadable ballots.

Voting Voters fill out their ballots, often with the help of software installed on their own computers.

Casting Election officials receive the completed ballots. As with distribution, different voting systems use different ballot casting mechanisms.

Tallying The tallying phase includes the remainder of the tasks that finalize the election. Counting votes and announcing the election outcome are common to almost every election system, though some include other tasks such as publishing information needed for audits.

Auditing Some elections will inevitably be disputed. In such cases, there is a final phase in which interested parties look for evidence that the election outcome is correct (or incorrect).

One major concern for Internet voting involves ballot integrity during the distribution, voting, and casting phases. For the election outcome to be correct, it is important that:

- the blank ballot that is received by and displayed to the voter match the ballot created and sent by the election officials;
- the computer used to fill out the ballot faithfully reports the intention of the voter; and
- the filled out ballot received by the election officials is the same as it was when the voter sent it.

Typical Internet communications involve not just the computers owned by the two parties communicating, but also many intermediary computers controlled by neither party. For example, an email you send from anywhere in the world will travel through multiple servers on its way to its destination. At any point, one of these servers could intercept and modify the email without your knowledge. A good election system needs to account for this, making it impossible for these intermediaries to intercept ballots for viewing or modification during transit.

Another concern is that most voters are not system administration experts, and many of their computers are compromised by malware. A compromised computer may corrupt the voting phase: even if the voter receives an unaltered ballot, malware may change the way the ballot is displayed or the way the vote is recorded before casting the ballot. It can be difficult to design a system that can resist this kind of attack without significantly reducing the usability of the system. Some systems, referred to as dual (or triple) channel systems, use alternative distribution mechanisms as cross-checks. For example, election officials may send a code by postal mail that a voter can use to check that a displayed ballot is correct.

As much as possible, Internet voting should be private and anonymous. It is essential that voters are free to vote the way they choose, and do not feel pressured to vote for a particular candidate or vote a particular way on any issue. The fewer people who know or can find out how a voter voted, the more comfortable the voter can feel about privacy.

At the same time, election systems must only record votes from people who are registered to vote, and must only record one vote from each voter. Election systems must balance the need to keep votes anonymous agsinst the need to ensure that a vote is coming from somebody who should to be able to vote.

One popular approach to this problem in existing systems is to require that each vote be tied to the voter who cast it long enough to decide whether to include the vote in the later tally or not. After it is decided to include the vote, the system records the vote and deletes the information about who cast it. This approach can work; however, audits of systems that take this approach have shown that it is easy to accidentally retain the connection between votes and voters longer than intended. This makes information much more widely visible than intended. It would be better if voters could be confident that the system does not store any connections between their votes and their identities. To accomplish this, the information a voter returns during the ballot casting phase must not include any personally identifying material.

This is a subtle, but crucial, distinction. The overall objective is for Internet voting systems to be correct, private, secure, and so forth. It is important for the people who develop these systems to verify that they are correct and take an active role in seeking out and eliminating defects in the system. However, verifiable Internet voting goes even farther: the system must be *visibly* correct. That is, voters using the system must be able to *check* that it is behaving correctly, without trusting that the system has no bugs or trusting that the system's behavior has not been influenced by third parties. It is especially difficult to prove to voters that their personal information has been deleted in order to protect their privacy, so voters may not want to provide their information in the first place.

This is why it is critical that an Internet voting system not only be correct, but *verifiably* correct. Verifiability is one of the central concerns of Internet voting, and is a critical part of the defense against the software bugs, security vulnerabilities, and sophisticated cybercrimes that history tells us are sure to occur.

The tallying process provides a particularly good example of the difference between correctness and verifiability. Voters certainly want the election system to count the votes correctly, but the goal of verifiability is to provide some *evidence* to voters that the election outcome is correct. Different systems attempt this in different ways. For example, they may allow voters to check that:

- their vote was included in the election outcome;
- the system is recording the content of their votes correctly; or
- the number of people that voted for a given candidate is accurately calculated (this must be proven without revealing any of the individual votes).

Meeting these verification objectives without violating anonymity and privacy is a balancing act. Each of these individual objectives contribute to a single top-level goal: *end-to-end verifiability*. "End-to-end" means that the whole election process produces a result that matches the intentions of the voters.

The verification objectives can be summarized with the catchphrase, "Cast as intended; recorded as cast; and counted as recorded."

- "Cast as intended" is the demand that casting use secure communications and other mechanisms to ensure that malware and outsiders cannot change the vote.

- "Recorded as cast" is the demand that the election system itself correctly interprets a vote.
- "Counted as recorded" is the demand that the tallying process be accurate.

These demands are subject not only to correctness but also to verifiability, so that voters can believe these properties hold even if they suspect that the systems, or the election officials, are corrupt.

3.2 Shortcomings and Expectations of E2E-VIV

As discussed in Chapter 2, several difficulties exist with current voting processes:

- voters with disabilities cannot vote unassisted;
- communication channels with remote voters are often slow and unreliable;
- vote tallying is labor-intensive and error-prone;
- election audits are costly; and
- there is little visibility into the election process, so voters (and, in some cases, even auditors) must trust the reports of election officials and voting hardware vendors on election outcomes and processes.

Internet voting may be able to alleviate some of these concerns. Voters with disabilities could potentially use their own familiar hardware, such as Braille displays, screen readers, sip-and-puff input devices, and others, to participate in the election. Internet communications are traditionally speedy (taking seconds rather than weeks) and relatively robust compared to overseas postal mail. In most systems, tallying is automated and fast. Auditing can still be a challenge, though it is expected that verifiable systems can make elections more transparent for this purpose. However, implementation of Internet voting has its own challenges. A system that properly addresses privacy and security concerns may be too complex to use and maintain or too costly to deploy.

3.3 E2E-VIV in Practice

Several practical voting systems have been developed based on the principles of E2E-VIV. This section describes some systems that communities have used in real elections or pilot tests.

3.3.1 RIES

RIES, the Rijnland Internet Election System [105], was first used in 2004 to support elections to the Rijnland water management board. RIES supplemented the system of postal voting already in use. Election officials used a later version of RIES to allow expatriate voters to participate in the Dutch parliamentary elections [94].

Before a RIES election, credentials—in the form of a very long number—and instructions are sent by postal mail to every voter.

During the election, each voter logs into the election website. The website includes a voting application written in JavaScript; the voting application is a "client-side" application, which means that it runs on the voter's computer and not on the election server. The client-side application processes the voter's authorization code and the public ID of the candidate to create an encrypted vote. The encrypted vote is then placed on a public online "bulletin board" that serves as a ballot box.

At the close of the polls, the election authority releases the final vote tallies along with a codebook containing the encryptions of all valid credentials and all candidate IDs.

The algorithms and protocols used by RIES are public, and each voter has access to all of the inputs and outputs. In principle, each voter may check the computations. This level of verifiability is weaker than the desired individual verifiability of E2E-VIV, but is nonetheless far stronger than conventional voting systems.

In 2006, the Organization for Security and Co-operation in Europe (OSCE) sent an election assessment team to observe the use of RIES. Their report [155] stated that they could not observe many critical security features of the system, and therefore could not be certain of their effectiveness. In 2008, the Eindhoven Institute for the Protection of Systems and Information (EiPSI) revealed several weaknesses in RIES [106]:

- the voter self-check procedure is quite complicated;
- the two-channel (mail and Internet) voting makes the system less transparent;
- too much power is given to the election administrator and the system's Internet host;
- issues arise when modifying the codebook due to a revoked ballot; and
- realistic ways to forge votes via cryptographic means (in particular, using hash collisions) exist.

One of the more important lessons learned through RIES is that when voter authorizations are distributed far in advance of the election, a mechanism must be provided that allows voters to obtain replacement credentials and invalidate lost credentials. This mechanism adds significant complexity to the system, and is a source of some of the problems reported in the OSCE and EiPSI reports.

Another feature of RIES, which has significant tradeoffs, is the ability to perform testing during the election: pre-invalidated test ballots are deliberately added to the bulletin board in order to test the network path from selected Internet clients to the server. While such testing can, in principle, increase confidence in the election integrity, in practice it opens the system to spoofing and denial of service[1] attacks. Moreover, the RIES implementation is aware when it is processing test ballots rather than real ballots, and all of the test ballots are typically voted identically from the same computer. This significantly limits the confidence provided by such testing, at the expense of increased system vulnerability.

Because of these critical reports, election officials discontinued plans to use RIES in the 2008 Dutch parliamentary elections, and the Netherlands banned Internet voting.

3.3.2 Prêt à Voter

In 2007, the University of Surrey in Victoria, Australia attempted to use the Prêt à Voter system [34, 42] in a student election [26]. This attempt was stopped prematurely, due to system malfunctions and an inability to open polling at all polling stations on time. The failure illustrated many of the pitfalls of adapting a research system to an actual election, such as a short timetable, a lack of clear requirements, and the need for rigorous implementation practices.

Prêt à Voter uses two-part paper ballots. Candidate names appear on one part and voting targets—the areas that voters must mark to cast their votes—appear with a ballot ID number or barcode on the other part. Typically, the two parts are printed as a single sheet with a perforation to divide the sheet after voting.

From the voter's perspective, the order of the candidate names on the ballot is random. The voter makes a mark next to the candidate name she chooses, separates the two parts of the ballot, and destroys the part containing the candidate names. She may take a copy of the voted part home for later verification.

For tabulation, a cryptographically secure mapping exists from the ballot ID numbers to the apparently-random orderings of the candidates. The system uses this mapping to decode the cast ballots into readable ballots with candidate names, while removing the associations between ballots and their ID numbers. The decoded ballots are then posted to a public online bulletin board.

Unvoted ballots may be audited before, during and after the election to ensure that the decoding of cast ballots is being correctly performed. Randomly selected stages in the decoding can be challenged to prove the integrity of the count, and any interested party can easily count the decoded ballots for verification.

[1] A *spoofing* attack deceives the target into believing that an Internet communication came from somewhere other than its actual source. A *denial of service* (DoS) attack is an attempt to make an Internet server unavailable to its intended users.

Individuals may also search for their voted ballot IDs on the bulletin board. This reveals the positions that were marked on that ballot but, crucially, does not show the corresponding candidate names. A voter can verify that the positions she marked at the polling place were correctly recorded by the system, but because she no longer has the part of the ballot linking candidate names to ballot positions, she cannot prove to anyone else how she voted.

3.3.3 Punchscan

Punchscan [162, 163] was used for the graduate student association elections of the University of Ottawa in 2007 [75]. It is likely the first E2E voting system with ballot privacy used in a binding election.

The election experience for a Punchscan voter is very similar to that of Prêt à Voter. Punchscan also uses a two-part paper ballot, but the two parts are overlaid one atop the other. The top part has candidate names and candidate numbers (or letters), and the bottom part has numbered (or lettered) voting targets; both parts have an identical serial number. The voting targets on the bottom part are visible through holes punched in the top part. The order of the voting targets for each race appears random to the voter.

The voter casts a vote by marking a choice with a bingo dauber—a thick marker used to fill in circles on bingo cards— and the two halves are separated. Either side can be cast as a ballot, since the bingo dauber marks both: one through the hole and the other around it. The uncast side is destroyed, and the voter may retain a copy of the cast side.

Anybody may inspect the public record of any cast ballot, exactly as with Prêt à Voter; this allows voters to verify that their receipts match the public record. It does not matter which half of the ballot a voter retains, since neither half by itself contains the necessary information to determine the vote. Like Prêt à Voter, Punchscan uses a cryptographically secure mapping from the ballot serial numbers to the apparently-random orderings of the candidate voting positions to decode the ballots for counting. Individual ballots may be audited to ensure that the decoding process is carried out correctly.

3.3.4 Scantegrity II

Scantegrity II (Invisible Ink) [44, 45] was used in the Takoma Park, Maryland municipal elections in 2009 [37]. It was also used in 2011 for in-person voting, along with Remotegrity (Section 3.3.5) for absentee voting. The 2009 Takoma Park election was the first use of an E2E-V system with ballot privacy in binding governmental elections.

Before the election, a *random seed* is generated and shared among election officials. It is very difficult to generate truly random sequences of numbers, so computers use programs called *pseudorandom number generators* to generate sequences of numbers that appear random. Pseudorandom number generators use data called a random seed when they start generating a sequence; every time the same generator is started with the same random seed, it generates the same sequence. The selection of good random seeds is very important in secure computing systems.

The sharing of the random seed is done using a *secret sharing scheme*. In a secret sharing scheme, a computer "splits" a piece of secret information—in this case, a random seed—into multiple parts and distributes these parts to multiple people. Combining some required number of the parts—this could be all the parts, or some smaller number, depending on the secret sharing scheme—reveals the secret information, but combining fewer than the required number of parts reveals nothing.

Using the random seed, the system generates a three-letter alphanumeric code for each choice on each printed ballot. It also generates additional tables so that interested parties can later confirm that the system computed the tally correctly.

During the election, the voter experience is nearly identical to that of conventional optical-scan paper ballots. When a voter marks a choice, the ink in the pen reacts with invisible ink on the paper to disclose the three-letter code in the marked voting target. The ballot ID number and the displayed code are posted to a public online bulletin board.

After the election, members of the public can verify the final tally using the bulletin board in a manner similar to that of Punchscan and Prêt à Voter. In addition to this public verification, individual voters can record their ballot ID numbers and the codes revealed from the invisible ink. They can then use the bulletin board to check that their ballots were tabulated. This information, however, is not sufficient to prove that they voted a particular way.

3.3.5 Remotegrity

Remotegrity [210], which was used for absentee voting alongside Scantegrity (Section 3.3.4) in the 2011 Takoma Park, Maryland municipal elections, is a *code voting* system. Code voting is a common scheme used in unsupervised remote voting systems. Voters enter a cryptographically-generated code corresponding to a candidate in order to vote for that candidate. Each voter gets a different set of codes, so external observers cannot learn which candidate a voter selected from the code the voter entered.

With Remotegrity, voters receive a code voting ballot and an authentication card in the mail. The codes on the ballot are covered by a lottery-style "scratch-off" field. The authentication card contains several authentication codes under scratch-off fields, a lock-in code under a scratch-off field, and an acknowledgment code. Both cards have serial numbers. Election officials can send each voter two ballots so that one can be used for auditing purposes.

To vote, a voter enters both serial numbers, the codes corresponding to her choices, and an authentication code obtained after scratching off a field chosen at random.

The voter can returns to the election website a few hours later to check if her codes are correctly represented, and to see if the election authority has posted her acknowledgment code next to her voting codes. This indicates to the voter that the election officials received valid codes for her ballot. She then scratches off the lock-in code and posts it on the website. This affirms to the election officials, observers and other voters that her vote is correctly represented on the website.

Among all the systems discussed here, this is the first one that asks the voter to take positive action to confirm that the vote was correctly posted.

Part of the security of Remotegrity comes from a separation between the computer that generates the authentication codes and voting codes (the "generator") and the computer that collects the votes (the "vote collector"). Because the generator and the vote collector share no information, the vote collector does not know what codes correspond to what candidates. It therefore has no information (other than codes) about how any voter voted and, because it has no information about what codes are valid, cannot change any cast votes. In addition, the vote collector has no information about acknowledgement codes; when a voter's correct acknowledgement code appears on the website, she knows that the election officials have received a valid code for her ballot because only the election officials could have posted the correct acknowledgement code.

The tally in Remotegrity is computed from the codes in a verifiable manner that corresponds to the code voting system used.

If a jurisdiction is concerned about using the Internet for remote voting, Remotegrity ballots can be mailed in, and voters can check for their codes on the election website to be assured that their vote correctly reached election officials.

3.3.6 Helios

Helios [8, 9] is a system developed for web-based E2E-V Internet voting. It was used for the election of a Belgian university president in March 2009 and has been adopted by numerous universities and associations since then, including the Association for Computing Machinery and the International Association for Cryptologic Research.

Before an election, officials input the email addresses of the voters who will be participating into the Helios system. The system emails each voter unique randomly-generated login information and the link to the election website.

During the election, a voter enters her choices on the website. After entering her choices, the voter has an option to "spoil" their ballot and request a new one. This invalidates the ballot, so that it is not counted in the final tally, but still allows the voter to verify that it was cast as intended. Spoiling a ballot is also a way to practice using the system and gain confidence in its accuracy. Once the voter casts a ballot that she does not declare to be spoiled, the system sends an email confirming the receipt of her vote, but does not include her choices in the email. At any time before the close of the election, the voter can repeat these steps and the new vote will replace the old vote.

After the election, Helios uses cryptographic techniques to tally the votes without revealing how individual voters voted [33, 194].

Helios does not require voter authentication until after the voter decides to cast a ballot, so any interested party may prepare and audit ballots. All cast ballots are posted in encrypted form on a public online bulletin board so that voters may check that their ballots have been correctly recorded. Similarly, after the polls close, the decryption and vote tally may be checked.

3.3.7 Norwegian System

Between 2011 and 2014, the Norwegian government ran a remote Internet voting trial [88] using a cryptographic protocol designed by Scytl, a commercial voting system vendor. The Norwegian system uses a combination of postal mail, the Internet, and SMS text messaging.

Before the election, the voter receives authorization codes to cast a ballot via postal mail. During the election, the voter uses a computer to cast an encrypted ballot. The voter can cast multiple ballots, but only the last ballot cast is counted. If a voter votes both on paper at a polling place and by Internet, the paper ballot overrides the Internet ballot. After casting a ballot, the voter receives a confirmation code offering a partial verification via an SMS text message.

Available descriptions of the Norwegian system are incomplete, so it is not possible to analyze the system in depth. However the system's claims with respect to voter privacy are weak: "If the voter's computer and the return code generator are both honest, the content of the voter's ballot remains private."[2] In addition, the receipt delivered to the voter proves only that the encrypted ballot was received as cast, not that it was counted as cast or that the encrypted vote matches the voter's intent.

The Norwegian system evolved between its first use in 2011 and 2013. Significant complexity was added in an attempt to assure voters that their ballots were stored as cast. In 2013, the Carter Center mounted a serious effort to observe the Norwegian system in action. Their report on the operation of the system and the problems they had observing it offers useful insight into the administration of E2E-V systems in general, as well as the particulars of the Norwegian system [38]. Scytl and the Norwegian government assert that this is an E2E-V system. However, if a voter's encryption software and the return code generator share information, they can lead her to believe that her vote was cast accurately even when it was not. As such, the Norwegian system's property of voter verifiability relies on the voting system software functioning properly. True E2E-V systems can not rely on the correctness of the voting system software for verifiability.

The Norwegian system also does not provide a proof of the tally, and is therefore not universally verifiable. If a voter casts multiple ballots to avoid coercion, she cannot verify which ballot was counted. Since the tally cannot be correct if the correct votes are not included in the count, the voting public also does not have the information to determine that only one vote was counted for each voter. For all these reasons, the Scytl software implementation used in the Norwegian elections is not considered an E2E-V system.

[2] This claim is part of the voting protocol description for the Norwegian system [88]. "Honest" means that the software on the computers in question has not been corrupted in an attempt to subvert the election.

3.3.8 Wombat

The Wombat in-person voting system [104] has been used for multiple pilot elections in Israel. A voter fills out her ballot on a touch-screen and receives a printout of both encrypted and unencrypted versions of her vote. She can then choose either to cast or to audit the encrypted vote. If she chooses to audit the vote, she may check that the vote was correctly encrypted. If she chooses to cast it, the unencrypted vote goes into the ballot box and the encrypted vote is posted on a public online bulletin board; the voter can take her copy of the encrypted vote home to verify that it was posted. The encrypted votes are tallied cryptographically, and the unencrypted votes in the ballot box can also be hand-counted.

3.3.9 DEMOS

DEMOS [65] is a code voting system where the voter is given a two-part coded ballot. She uses one part for auditing and the other to vote. Associated with each choice on the ballot is a vote code; the vote code includes the encryption of the vote, which is entered in the voting machine by the voter, and a receipt code, which the voter does not enter, but which is posted online next to the vote code.

The voter can check the receipt to ensure her vote reached the election authorities. The ballot also has a QR code containing all the information on the ballot, which can be scanned by the voter if she prefers not to manually enter the vote code. Once the ballot is entirely represented on the computer, the voter can make her choices. If the voter scans the QR code, the scanning computer knows how she voted. The vote codes are encryptions of the votes, and a verifiable tally is obtained in a standard manner.

A pilot study of DEMOS was carried out during the 2014 European Elections in Greece.

3.4 Limitations and Tradeoffs of Existing E2E Systems

E2E systems inherit many of the limitations of traditional voting systems. Reliability of equipment, reliance on procedure, trust in insiders, and accessibility are all problems with traditional in-person voting systems. For remote systems, the integrity of postal systems, turnaround time for mailed materials, access to Internet or fax technology, and reliability of Internet servers are all well-documented obstacles to voting.

Existing E2E systems mitigate some of these limitations, but have other limitations of their own. In this section, we examine the limitations of E2E systems with a particular focus on those that are unique to or exacerbated by E2E characteristics.

3.4.1 Vote Secrecy

Systems like Prêt à Voter (Section 3.3.2) and Punchscan (Section 3.3.3) rely on a randomized candidate order or a code on printed ballots to ensure vote secrecy. Voted ballots must appear on a public online bulletin board in order to verify the election results. To protect secrecy, only the selected position or code is visible on the final ballot along with a ballot ID.

If an insider is able to review the printed ballots before the election, they can record how the candidate positions are arranged for each ballot ID. The insider could then violate secrecy by identifying the candidates marked on the voted ballots [34].

Recent work on Prêt à Voter recommends printing ballots on demand at polling places in order to limit this possibility [171]. However, printing on demand introduces additional problems and expense compared to centralized printing. More printing equipment is required at each polling place, and that equipment can break or be difficult to operate. The printing equipment must also have some way of communicating with the rest of the election infrastructure to ensure that it has the correct cryptographic seeds for generating new ballots.

Scantegrity II (Section 3.3.4) uses invisible ink to hide the vote codes on unvoted ballots, and Remotegrity (Section 3.3.5) can use scratch-off fields to hide vote codes and other information required to cast a ballot. These techniques limit the opportunity for insiders to learn secrecy-compromising information without being detected through the presence of a marked or damaged ballot.

Even with techniques to mitigate insider foreknowledge of the ballots, secrecy can still depend on voters and poll workers correctly following procedures. An important aspect of secrecy is *receipt freedom*: voting systems should not provide receipts (akin to purchase receipts) that allow voters to prove how they voted, because these enable both coercion and vote selling. However, some E2E systems are only receipt free because of procedure; for example, a voter can leave the polling place with a complete Prêt à Voter ballot, failing to shred the half with the candidate order. With both halves of the ballot, she can prove how she voted.

RIES (Section 3.3.1) makes a deliberate secrecy tradeoff by weakening the receipt freedom requirement in exchange for providing universal verifiability and a degree of individual verifiability. The results of an entire election can be independently audited using only the information that is publicly available after the election. However, if a voter discloses her credential or her encrypted vote, the same public information may be used to violate ballot secrecy. The developers of RIES judged this violation to be no more severe than the threats to ballot secrecy inherent in postal voting, and therefore worth accepting for the benefit to verifiability.

3.4.2 Ballot Stuffing

As when ensuring vote secrecy, many E2E systems depend on correct procedures to defend against ballot stuffing. For example, during the University of Ottawa elections using Punchscan, more ballots were cast than voters recorded in the pollbook. In this case, ballot stuffing can be caught after the fact by poll workers, but it is not an inherently verifiable property of the system and it requires trust in the accuracy of the poll workers.

In the Helios system officials can register voters in the system by email address, so there is limited protection against insider ballot stuffing. Helios relies on individual voters verifying their votes. However, while an interested party may verify the tally for the entire election by checking that the collection of encrypted votes is counted correctly, no provision exists to determine if votes were fraudulently cast on behalf of register voters who did not vote [175].

A pre-election step that publicly publishes tables of valid ballot IDs can help mitigate this problem, but also creates others. All votes in the final tally, even though they are anonymized, can be traced back to before the election began and cross-checked with voter registration rolls. However, having a fixed set of ballot IDs can make it harder to replace lost, stolen, or spoiled ballots, or to provide for late or same-day voter registration.

3.4.3 Dispute Resolution

E2E systems provide voter verifiability: they enable a voter to determine if her vote was accurately recorded. However, when the vote is not accurately recorded, not all E2E systems provide the voter with evidence that can be used to convince an independent party of the problem. This creates a vulnerability that dishonest voters could exploit. For example, a dishonest voter could raise doubts about the legitimacy of the election by incorrectly claiming that their vote is not accurately recorded.

An E2E system with dispute resolution enables a voter to present evidence to support claims of election fraud. It also enables a third party to resolve a dispute between a voter who claims her vote is inaccurately recorded and the voting system that claims it is accurate.

All cryptographic protocols in the academic literature that provide dispute resolution[3] require either paper or a second electronic method, such as a smartphone, in addition to the machine the voter uses. This is true whether the voter votes in a polling booth or remotely. Therefore, any voting system with dispute resolution should use either paper or a second electronic method.

[3] This includes only the protocols intended for use by voters who vote from untrusted machines.

3.4.4 Infrastructure and Equipment

Election equipment can fail in practice. An E2E system must be resilient to failures while not giving up E2E properties. A system that lacks recovery mechanisms is not robust; it is only as strong as its weakest recovery mechanism. For example, if a remote voting website fails and election officials resort to accepting voted ballots by email, E2E guarantees are lost for all the emailed ballots.

In addition to being more sensitive to failures, verifiable election systems often require more sophisticated equipment than traditional systems. For in-person voting, a verifiable system might require ballot printers for on-demand printing, a high-quality shredder for two-part ballots, and more sophisticated assistive devices. This adds cost and increases poll worker training requirements.

Many E2E systems post encrypted ballot information to a public online bulletin board during the election. Most of these systems assume that the bulletin board is *write-only*. On a write-only bulletin board, information can never be removed or changed once it has been posted. Moreover, the order in which items are listed on the bulletin board cannot be changed.

In order to update such an online bulletin board in real time, these E2E systems are distributed systems; they can be networked via traditional means, or information transfer among machines can be carried out manually by election officials using USB flash drives or similar devices. Depending on the networking scheme, this can leave the system vulnerable to various security threats and denial of service attacks. At least part of the system must be connected to the public Internet, which increases the possibilities for malicious attacks.

Many systems allow voters to use their own computers to vote. While convenient for voters, this can cause problems because election officials cannot control the voting environment on voters' computers. Threats include:

- malware on the voter's computer, which can undermine security;
- incompatibilities that arise because of the voter's operating system or web browser versions; and
- compromised network infrastructure between the voter and the central election system, which can allow votes to be intercepted or changed by third parties.

In general, it is widely recognized that any E2E-VIV system must be *software independent* [169]. A voting system is software independent if an undetected change or error in its software cannot cause an undetectable change or error in an election outcome. A voting system is *strongly software independent* if, in addition to being software independent, it enables a detected change or error in an election to be corrected without re-running the entire election. A requirement contained in this report insists that any future E2E-VIV system must be strongly software independent.

3.4.5 Usability

Traditional election systems struggle with usability. Most often, however, the problems can be addressed with special attention to user interface design. Verifiable systems add steps and complexity to the voting process, presenting usability complications that researchers have not fully studied or overcome. For example, marking a ballot is more complex with code voting, as with Remotegrity, and with position or shape matching, as in Prêt à Voter and Punchscan.

Individual vote verification, not possible with traditional voting systems, is an entirely new process that voters must master in order to take full advantage of the positive E2E-V properties. Many E2E systems have attempted to reduce the additional effort for voters who are not interested in verifying their cast votes.

In 2014, a team of researchers from Rice University studied the usability of three E2E-V systems: Helios, Prêt à Voter, and Scantegrity II [5]. They sought to measure usability by examining effectiveness, efficiency, and satisfaction. Their results show that these systems broadly fail in the area of usability, even for typical voters who are not interested in performing additional verification steps.

The researchers found that a significant number of voters failed to cast a ballot with each of these three systems, rendering them ineffective. Many of those voters thought they had successfully cast a ballot, only to discover that the process had failed them. In a real election, those voters would have left the voting process unfinished without realizing that there was any problem. Traditional voting systems have success rates close to 100% [35].

The E2E-V also lacked efficiency. They all required significantly more time for voting. It took almost twice as long to vote with the E2E-V systems as with a traditional system.

The usability aspects of an election system are crucial. An election system must not disenfranchise voters; on the contrary, it should make voting easy and assist voters in completing the process. Voters must also be confident in the election results. The Rice study shows that adding E2E guarantees can add verifiability at the expense of usability, resulting in a voting system that is unusable by voters. Some researchers do not agree with the Rice study [137]; however, it is an important reference point for E2E-V development.

3.4.6 Accessibility

Many E2E systems have requirements for voters' abilities. For example, a typical voter can see where to make a mark for a particular candidate on a Punchscan ballot, but a sight impaired voter cannot do the same without help. In addition to obstacles to marking a ballot, some types of E2E systems lack features to allow disabled voters to participate in individual vote verification without help. Information required for verification is frequently delivered through a paper receipt, an invisible ink code, or receipt data that a voter must write down.

Researchers have proposed accessible verification protocols that protect vote secrecy and allow voters to participate in individual vote verification [43]. These protocols require using accessible devices with an audio, sip-puff, or switch interface to read and mark the unencrypted ballot. The voter must trust that any special device she is using will not create a record of how she voted, which would violate vote secrecy. The device must also represent the ballot faithfully to the voter so that votes are recorded as intended.

Requiring trust in accessible devices is not unique to E2E systems [170]. In non-E2E systems, voters are required to trust many aspects of the election. Because voters must trust the chain of custody of ballots, the integrity of poll workers, and the outcomes of any audits, having to trust an accessible device is a relatively small concession to make in an already-flawed system.

On the other hand, a well-designed E2E system requires a much smaller base of trust for voters to be confident in the election results. The additional requirement of trusting an accessible device is not a flaw of the E2E system; it is part of the nature of using accessible devices. This vulnerability may be lessened by allowing voters to use multiple accessible devices, or by requiring all voters—both those who need help and those who don't—to use similar accessible devices or a universal interface. This ensures that multiple users test the accessible devices or interfaces.

3.4.7 Social and Political

New election systems face a difficult problem. In order to be adopted in large-scale elections, they must have a successful track record. Building a record of success, given the limited financial and human resources available for early small-scale pilot programs, is a major challenge. With limited resources, election officials may be forced to compromise on some aspects of the election systems they implement. This increases the risk that problems with equipment, software, and support will undermine confidence in the system.

Public confidence in election systems in general, and E2E systems in particular, is fragile. When election systems fail during an election or are revealed to have substantial integrity issues, people may believe that all similar systems are flawed. It does not matter if the E2E systems are very different from each other, or if the E2E systems provide guarantees. Failure of an older E2E system can cause the public to reject a new system.

In 2009, there was a hacking demonstration on electronic voting machines used in previous elections [86]. As a result, the Federal Constitutional Court of Germany decided that electronic systems may only be used in elections if "the result can be examined reliably and without any specialist knowledge of the subject." In practice, this is a standard that E2E systems have not been able to meet [35]. Similarly, after reports critical of RIES were released in the Netherlands, a popular movement successfully advocated for a ban on Internet voting in that country as well.

Many people are aware of computer security concerns and vulnerabilities, and are worried by them. Hacking, malicious code, and large amounts of personal data being stolen electronically are almost weekly news. These concerns rightly make people wary of any system with a computerized component, even if the Internet is not involved. The challenge for E2E systems is to overcome this broader skepticism by demonstrating integrity in a way that anyone can understand without making it more difficult to vote.

Chapter 4

Required Properties of E2E Systems

In August 2010, the U.S. Election Assistance Commission issued a set of testing requirements for overseas and military remote electronic voting system pilot projects [198]. However, the EAC requirements have some serious shortcomings. Many of the requirements are arbitrary, inappropriate, or invalid: some set maximum system error rates using specific numbers without justifying the numbers; some set unrealistic limits on the accuracy of computer hardware; and some prohibit developers from programming in ways that are widely used when implementing highly reliable software systems.

If these issues were addressed, the EAC requirements could serve as a solid set of requirements for remote electronic voting systems. However, they are not strong enough to guarantee end-to-end verifiability, which is essential when considering Internet voting systems for use in real elections.

A set of E2E-VIV requirements, which significantly overlaps with the EAC requirements, can be broadly divided into two groups: *technical requirements* and *non-functional requirements*. Technical requirements are those that can be directly addressed by the design and implementation of the system, such as authentication requirements for voters and election officials. Non-functional requirements are those that are imposed on the system by external entities or where the system depends on external behaviors outside its control, such as specific election certification guidelines and operational procedures. The technical and non-functional requirement groups can be further divided into several categories, and Figure 4.1 gives a high-level overview of these.

The following is a high-level description of the E2E-VIV requirement categories and many of the requirements; the full set of E2E-VIV requirements expressed in the Business Object Notation is available as a separate document [126].

4.1 Technical Requirements

There are ten categories of technical requirements for E2E-VIV systems: functional, accessibility, usability, security, authentication, auditing, system operational, reliability, interoperability, and certification.

4.1.1 Functional

The functional requirements of an E2E-VIV system deal primarily with the casting and recording of ballots and associated voter records. One such requirement is that recorded ballots and voters listed as having voted must correspond with each other; a ballot cannot be recorded without a voter casting it, and a voter cannot be listed as having voted without casting a ballot. If the system tells a voter that her ballot has been successfully cast, the system must correctly retain the record that she has voted. The system must keep a voter's cast ballot information even if servers fail.

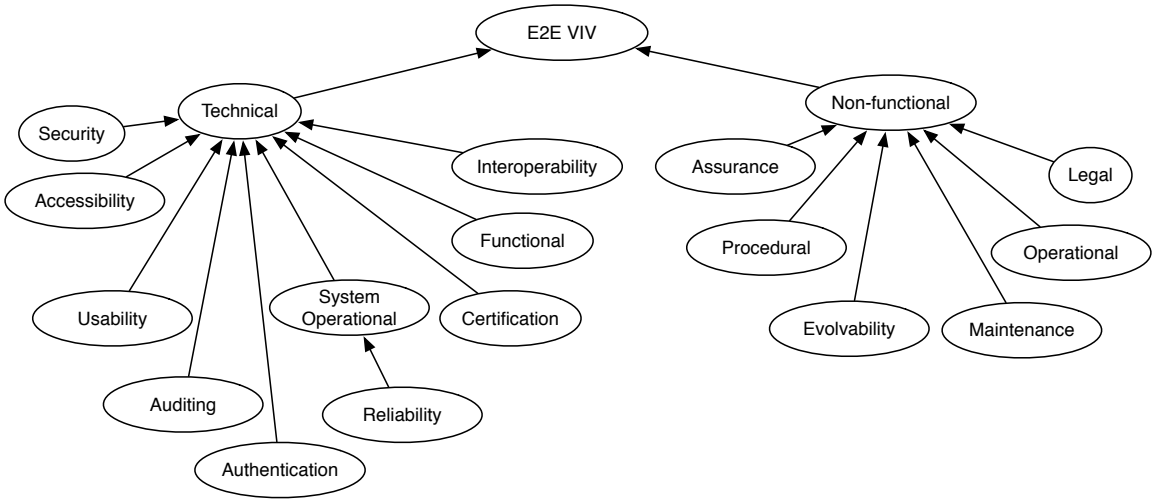

Figure 4.1: The hierarchy of requirements for E2E-VIV systems.

Another functional requirement is *receipt freedom*: it must be impossible for a voter to prove to anybody any information regarding how she voted, beyond what someone can mathematically deduced from the final distribution of votes. If a referendum passes with 100% of the vote, for example, there is no way to hide the fact that every voter approved of the referendum. If the result is mixed, it must be impossible for any individual voter to prove how she voted. This must be true even when the voter can create digital evidence of her actions by, for example, video recording the ballot casting process or photographing a completed ballot.

No usable E2E-VIV protocol in existing scientific literature has receipt freedom when the voting computer is untrusted. Since the process of casting any individual ballot can be recorded, a protocol with receipt freedom must allow the voter to vote multiple times but only count the last ballot a voter casts. However, it must also ensure that an observer watching any recorded ballot casting process can independently verify that the ballot was cast. This must be ensured even when the voting computer is not trusted to follow protocol, and in a manner that is easy to use and accessible. This represents a significant research challenge for E2E-VIV protocol designers.

In some elections, voters are allowed to cast multiple ballots with only the last cast ballot counting toward the final election tally. In others, voters are prohibited from casting multiple ballots. The system must accommodate both of these election formats.

Maintaining voter anonymity is critical. It must be impossible after the election to reconstruct a link between a cast ballot and any identifying information about the voter who cast it. However, in systems that support the casting of multiple ballots, it is important to maintain links between voters and their ballots *during* the election to ensure that later ballots replace earlier ballots. To balance these concerns, any link between a ballot and the voter who cast it must be irrevocably broken once it is determined that the ballot will be counted toward the final tally.

4.1.2 Usability

The usability of an E2E-VIV system is critical to its successful adoption and use. Since user experience is so important, many of the requirements of the system have some relation to usability even though they may be categorized under other headings. There are, however, two requirements that are exclusively related to the usability of the system with respect to vote casting and a general usability requirement that applies to the system as a whole.

If a voter receives a final vote confirmation from the system, such as “Thank you for voting!”, the ballot casting process must be complete and the system must have recorded the vote. This allows voters to be certain that their ballots have been recorded and will be counted.

If a voter is uncertain whether or not the system recorded her ballot—for example, she clicked a "submit" button but never got a response from the system—she must be able to vote again.

Researchers must perform usability testing on any E2E-VIV system before it is deployed. The reports of the usability testing must be made public, and the system must achieve satisfactory test results before election officials use it in a real election.

4.1.3 Accessibility

Accessibility—the property of being usable by and useful to voters with disabilities—is one of the main goals of an E2E-VIV system. It is closely related to usability, but there are several requirements associated specifically with accessibility that go beyond typical usability requirements.

Users must be involved in the design of the system to identify accessibility problems at each stage of the development process. Developers must consider the system's compatibility with existing technologies designed to help individuals with disabilities. The system should be developed in a way that allows people to use accessible devices such as switches, eye trackers and screen readers in addition to keyboards, mice and touchscreens. The system should present voting options that are optimized for voters' needs by using alternative display fonts, audio representations, braille representations, and other representations as appropriate.

Developers must take all possible measures to ensure that as many voters as possible can use the system. Election officials must provide access to alternative methods of voting for those voters who cannot use the system.

Researchers must perform accessibility testing in addition to mandatory usability testing. The reports of the accessibility testing must be made public, and the system must achieve satisfactory test results before election officials use it in a real election.

4.1.4 Security and Authentication

Security and authentication are closely related. Together, they represent the broadest set of technical requirements. These include requirements on the E2E-VIV system itself, such as data storage and communications, and requirements on the voting and counting processes that the system enables, including voter authorization, voter privacy, vote integrity, and tally accuracy.

Developers must ensure data integrity throughout the system. No data can be permanently lost if the system breaks down or experiences a fault. The system must maintain the integrity of the voters' register, lists of candidates, ballot information, cast ballots, and other critical information. It must authenticate the original source(s) of all information and keep track of where the information came from. All data communications within the system must have associated integrity checks.

System equipment under the control of election officials must be protected against influences that could modify election results. The integrity of the election results must not depend, in any way, upon the security of system equipment that election officials do not control. The system must perform regular "health checks" to ensure that it is maintaining data integrity, all its components are operating in accordance with their specifications, and all system services are available.

Accurate timing information is critical to security, both to provide evidence of compliance with applicable regulations and to detect attacks on and potential breaches of the system. The system must therefore maintain reliable synchronized time sources, with sufficient accuracy to maintain timing data for audit information, election observation data, and time limits for various aspects of the election process. Using the timing information stored by the system, it must be possible to determine whether nominations (and, if required, the candidate's or election officials' acceptance), voter registration, and vote casting occurred within the time limits for those actions.

Authentication and authorization are also important aspects of security. The system must ensure that each individual can be identified uniquely, so that there is no possibility of mistaking one individual for another. The system must also maintain the privacy of individuals, by ensuring that all personally identifiable data is kept confidential as far as legal requirements of the electoral jurisdiction allow. The system must allow access to each of its services only to authorized users; for example, only election officials may be allowed to load ballot information into the system.

The means used to gain access to the system must, as far as possible, protect authentication secrets (passwords, one-time access codes, biometrics, etc.) so that unauthorized entities cannot acquire them. People may not use third party authentication mechanisms, such as existing Facebook, Google and Twitter accounts, to access the system.

Any potential breach of any public or commercial database, such as a credit card database or the Social Security database, must not affect the security of the voting system's authentication mechanism. An attacker should not be able to impersonate a voter even if the entire database used for authentication in the system is compromised. Individual authentication secrets themselves must be changeable or revocable at any time, at the individual voter's or election officials' request. The system should require all individuals to change their authentication secrets at least at least once in every election cycle.

The system must allow only eligible voters to cast ballots, and must ensure that each voter only casts the appropriate number of ballots. A voter must be able to verify that the system has presented her with an authentic ballot and, in the case of remote voting, that she has a secure connection to an official server.

The system must preserve the privacy and integrity of the vote, end-to-end, to the maximum extent possible. Individual voters may not waive the privacy of their votes. In the case of remote voting, the system must preserve vote privacy and integrity even if the voter's computer contains malicious code (corrupted client software, key logging software or devices, etc.). Any client software used in remote voting must not send data to any Internet host except those associated with the E2E-VIV system or provide any information to third parties (such as Facebook or Twitter) regarding the act of voting. The system must destroy any residual information that could be used to discover a voter's choices after a ballot has been cast. If a voter uses a computer outside the control of election officials to cast her vote, she must be provided with instructions for destroying the residual information on that computer.

The system must accurately count the votes, and the counting process must be reproducible. The system must also maintain the availability and integrity of all information used to generate the final tally and all information regarding the counting process itself for as long as required. Vote tabulation must be *strongly software independent*; it must be possible to detect any compromise of the election system software that causes a change in the tally, and to reconstruct a correct tally from some record in the event of such a compromise.

A deployed E2E-VIV system will likely be an attractive target for highly capable adversaries who wish to influence election results or to disrupt election processes. System designers and testers must assume that an adversary has a budget of US$10 per voter per election that they can apply toward any subset of votes or voters of they choose. This means that designers and testers of an E2E-VIV system for use in a U.S. presidential election must assume that an adversary has a budget of approximately US$1,300,000,000.

Election officials shall have overall responsibility for compliance with these security requirements, and independent bodies will assess compliance as appropriate.

4.1.5 Auditing

The ability to perform comprehensive audits of system activity is one of the important distinguishing aspects of an E2E-VIV system. compared to other voting systems. In addition to those security requirements (such as the tracking of accurate timing information) that touch on auditing, several system requirements relate specifically to auditing.

Developers must design and implement audit features as part of the E2E-VIV system from the beginning; they cannot be added as an afterthought to an existing system. Developers must implement audit and monitoring capabilities into all levels of the system, from low-level communications among individual computers to high-level interactions with election officials. The system must keep audit logs of all activity relevant to the conduct and outcome of the election. These logs must be locked, incapable of being modified by anyone, once they are written. They must also be as complete as possible without violating voter privacy.

The audit system must actively report potential issues and threats, rather than merely serving as a passive repository of system logs. It must record at least the following events and actions with accurate timing information:

- all voting-related information, including the number of eligible voters and votes cast, the number of invalid votes, count and recount results, etc.;
- any detected attacks on the operation of the system or its communication infrastructure; and
- any system failures, malfunctions, or other detected threats to proper system operation.

The audit system must provide sufficient information to election observers in real time, and after the election's conclusion, to verify that the election is carried out in accordance with applicable law.

The audit system must also be able to:

- cross-check and verify the correct operation of the voting system and the accuracy of the election results;
- detect voter fraud; and
- prove that all counted votes are legitimate and that all ballots have been counted.

In situations where the system cannot verify the legitimacy of all the votes, it must be able to report how many ballots may be affected.

If a tradeoff must be made between maintaining voter privacy and identifying the perpetrators of fraud, the system must resolve that tradeoff in favor of voter privacy.

For voters to trust an E2E-VIV system, its auditability must extend to its own source code as well as the activities it performs during an election. Developers must publish the E2E-VIV system software and any official monitoring and auditing applications in source form. They must include full documentation, instructions for building and running the software, and a digital signature as a proof of authenticity.

4.1.6 System Operational

System operational requirements ensure that the system is configured, updated, and run in a transparent, accountable way. One important requirement is that election officials must publish manifests of the system used to run any election. These manifests must include details of the software and versions they used, the dates they installed the software, and brief descriptions of the software's functionality. Well-defined procedures must exist to update the manifests to reflect changes to the installed software and to check the installed software against the manifests to detect tampering. Election officials must follow these procedures before every election period, and must also check all equipment and approve it for use.

During an election period, election officials must keep key equipment in a guarded, secure area at all times. They must have a contingency plan for system failures, including provisions for backup systems. The backup systems must conform to the same standards and requirements as the systems they replace. In addition, election officials must make sufficient arrangements to backup data; backup systems must be continuously monitored, and backup data must always be available during the election. Election staff must be ready to intervene rapidly, according to well-defined response procedures, if problems occur during an election.

The election officials and election system vendors must be accountable for the performance of the system. To ensure this, election staff must prepare a report after every election containing every software manifest change and every violation of data security, system security, physical security or control procedures that occurred during the election period. This report must be published within a reasonable amount of time after the election.

4.1.7 Reliability

E2E-VIV systems must satisfy strict reliability requirements that ensure their behavior is reasonable, both under normal conditions and while under attack.

In general, the back-end (i.e., non-voter-facing) components of the system must be able to run continuously, for at least a week, at the highest rate that election officials expect voters to participate. Multiple actual tests of mock elections must be run to demonstrate that the system satisfies this requirement. It applies only to normal operation, not while the system is under attack.

The system must also be highly available during the election period; voters should be able to access it 99.9% of the time. It must be able to recover from any failure, other than a regional natural disaster or malicious attack, in less than 10 minutes. This must be demonstrated by inducing failures in actual mock election situations, for example by unexpectedly unplugging servers or disconnecting storage devices. In order to ensure the 10-minute maximum recovery time, all critical parts of the system must have redundant backup systems that can take over if failures occur.

An E2E-VIV system is likely to be a tempting target for distributed denial of service (DDoS) attacks.[1] It must be able to continue correct operation during a sustained DDoS attack at a specified level (that is, a specified number of machines performing the attack or a specified amount of data being used in the attack), without slowing down by more than a specified amount during the attack. The specified attack level and acceptable slowdown will vary among election types. For example, a system running a national election must be able to resist a significantly higher level of attack than a system running a county election. Security experts' initial suggestions for the thresholds for a national election are that the system must continue operating correctly under a DDoS attack at a level of 100 gigabits per second, with no more than a 15 second slowdown.

The network configuration used during an election must show that the system can survive DDoS attacks and continue an acceptable level of operation. Election officials should re-evaluate the network configuration every election cycle to keep pace with advancing attack technology.

4.1.8 Interoperability

E2E-VIV systems must use open, rather than proprietary, data and communication standards for interoperability among their various components and services. Whenever possible, developers should use the Election Markup Language (EML), or a similar standard ratified by an international standards body, for data interchange and configuration within the system. The standards used within the system should allow for localization of election data, such as translation to different languages, where required.

Election officials must publish the log data for the system, and documentation describing its meaning and format, so that anybody can download, inspect, and publish concerns based on the system logs.

4.1.9 Certification

In order to provide sufficient evidence for certification of an E2E-VIV system, each functional requirement must have an associated set of automated tests. Election officials must be able to run these tests on demand. Test results should be unambiguous and easy to understand.

To the extent possible, system developers must provide formal proofs of correctness and security for the communication and cryptographic protocols implemented by the system.

[1] A distributed denial of service (DDoS) attack is an attempt to make an Internet server unavailable to its intended users by attacking the server from many other systems on the Internet at the same time.

4.2 Non-functional Requirements

There are five categories of non-functional requirements for E2E-VIV systems: operational, procedural, legal, assurance, and maintenance/evolvability.

4.2.1 Operational

The operational requirements on E2E-VIV systems deal with several distinct issues: voter assistance, election and registration timing, voter registration, candidate nominations and lists, receipt freedom, voter assistance, election integrity, and openness.

Voter Assistance Election officials must inform voters, in clear and simple language, how electronic voting will be organized and what steps voters will need to take in order to participate and vote electronically. Election officials must make support and guidance on voting procedures available to all voters. For remote voting, support and guidance must be available through a different, widely-available communication channel (such as a dedicated phone number) in addition to being available on the Internet. Voters must receive clear guidance on hardware platforms, operating systems, browsers, browser plug-ins, other applications that the E2E-VIV system requires. They must also be told what what common components, plugins, or other software, such as pop-up blockers and script blockers, may interfere with voting. Voters must receive clear guidance about configuration choices they can make to more strongly protect their privacy, such as:

- disabling cookies and browser history logging;
- running privacy-protecting browser plugins;
- voting from temporary virtual machines;
- logging out of social networks; and
- disabling non-election-related Internet communications.

Election and Registration Timing In any election carried out using an E2E-VIV system, legal provisions in the election jurisdiction must state clear timetables concerning all stages of the election. The period during which a vote may be cast electronically must not begin before election officials notify the public of the election. In jurisdictions that allow remote electronic voting, the voting period must be defined and made known to the public well in advance of its start. In jurisdictions where remote voting takes place concurrently with voting at supervised polling stations, remote voting should not be allowed after the period for supervised voting has ended.

Voter Registration An E2E-VIV system must have a publicly accessible voters' register that election officials regularly update. Each voter must be able to check that her information as recorded on the register is accurate, and must be able to request corrections. Election officials must authenticate all modifications to the voters' register.

Candidate Nominations and Lists On any electronic ballot, all voting options must be presented equally. An electronic ballot must not contain any distinguishing fonts, sizes, styles, or other embellishments that could cause a voter to think that one or more of the voting options are preferred. The ballot must not contain any information about the voting options, such as biographical information about candidates or interpretations of and statements about ballot initiatives. Only the information required for casting the vote or required by law (for example, candidate party affiliation) must be on the ballot. The system must not display any messages that may influence voters' choices. If additional information about voting options is available from an electronic voting site as part of an E2E-VIV system, it must be separate from the actual electronic ballot and presented without bias.

Receipt Freedom E2E-VIV systems must exhibit receipt freedom, which was described earlier as part of the functional requirements. Operationally, receipt freedom has two different meanings depending upon whether the voting apparatus is supervised (in a polling place) or unsupervised (as is the case in most remote voting systems).

In a supervised environment, voting information—images, sounds, etc.—should disappear from the voting apparatus as soon as the voter casts a vote. When the system provides paper proof of an electronic vote at a polling place, the voter must not be allowed to show it to any other person or remove it from the polling place.

In an unsupervised environment, the situation is different. An adversary or coercer can digitally record the voting process, or voters can record themselves with the intention of selling their votes. Third parties must not be able to use such recordings to prove, either during or after the election, that the votes shown in the recordings are counted in the final tally.

Election Integrity Developers will likely make E2E-VIV systems available for testing by voters and election officials, both before and during elections. To preserve election integrity they must indicate clearly, before the final casting of any ballot, whether the ballot is part of a real election or part of a test. If a test occurs simultaneously with a real election, the system should direct individuals casting test ballots to the appropriate voting channel so they can cast real ballots.

An E2E-VIV system must not disclose preliminary results to anyone, including election officials, until after the system has stopped accepting electronic ballots. The system must not disclose tally information to the public until after the end of the voting period, including all polling station voting. The system should perform any decoding required for the counting of the votes as soon as practicable after the end of the voting period. Election officials must be able to participate in, and observers must be able to observe, the counting process. The system must keep a record of the counting process, including timing information and identifying information for everybody involved in the counting process. If any irregularity affects the integrity of votes, the system must record that the affected votes had their integrity violated. The effect of integrity violations on the election results will vary depending on the legal provisions of the involved jurisdictions.

Openness Any deployed E2E-VIV system must function correctly as an open system, where large parts—specifically, any remote client hardware and software—are unknown, unsecured, uncertified, and completely out of the control of election officials. Researchers must be able to audit the system to the extent possible given this requirement. Developers and election officials should be able to apply the conclusions drawn from the audit process when developing systems and procedures for future elections.

4.2.2 Procedural

Successful deployment of E2E-VIV systems requires certain procedures relating to provisioning, certification, maintenance, availability, and use. Because such systems are critical pieces of public infrastructure, information about their functioning must be publicly available. Information about the specific components of a system must be disclosed, at least to the relevant election officials, as required for verification and certification purposes. Before an E2E-VIV system is introducted, at appropriate intervals after its introduction, and whenever any changes are made to the system, electoral officials must call upon an independent body to verify that the system is working correctly. The independent body must also verify that election officials have taken all necessary security measures.

After introducing an E2E-VIV system, election officials must take steps to ensure that voters undesrtand its use and have confidence in the system. These steps may include outreach, practice elections, and any other measures to educate voters. In particular, election officials must give voters an opportunity to practice any new electronic ballot casting method before, and separately from, the time voters cast electronic ballots during a real election.

Election officials must take steps to ensure the reliability and security of the E2E-VIV system. For example, they must guard equipment and provide suitable reliable power supplies. They must make every effort to avoid the possibility of fraud or unauthorized intervention during the voting process. Election officials must be satisfied that the E2E-VIV system is genuine, and is operating properly, before using it to conduct a real election.

Only election officials or individuals appointed by them should have access to the central infrastructure, the servers, and the election data. Election officials should establish clear rules for such appointments. Critical technical activities must be carried out by teams of at least two people, and the composition of these teams must be changed regularly. As far as possible, critical technical activities should take place outside of election periods.

To the extent permitted by law, election officials must allow observers to watch and comment on the conduct and results of any election carried out using an E2E-VIV system. During an election period, any authorized intervention affecting the system must be monitored by both election officials and election observers.

The system must maintain the availability, integrity, and confidentiality of the votes. It must also keep the votes sealed until the counting process begins. Any votes stored or communicated outside controlled environments must be encrypted. Recounts must be possible, and any features of the system that may influence the correctness of the result must be verifiable. The system must also support partial or complete re-runs of elections.

Election officials must establish clear technical and legal procedures to follow if voters can prove that the system did not accurately receive or count their votes. They must also establish procedures to follow if the official election verification application does not verify that the results of the Internet portion of the election are correct.

4.2.3 Legal

Legal requirements arise primarily from the application of existing law to E2E-VIV systems. These include requirements on accessibility and availability; on the counting of votes, number of votes per voter, and anonymity of votes; and on restrictions with respect to reverse engineering or testing of E2E-VIV systems.

To comply with accessibility and availability requirements, an E2E-VIV system must be understandable and easy for voters to use. Registration requirements for electronic voting must impede voter participation. E2E-VIV systems should be designed, as much as possible, to maximize the opportunities they provide for voters with disabilities. Unless remote electronic voting channels are universally accessible, they must be used only as an additional and optional means of voting beyond polling places or more traditional remote voting methods.

In all jurisdictions—those that use an E2E-VIV system exclusively and those that combine electronic and traditional voting systems—election officials must ensure that only one vote by each voter is counted. In jurisdictions that combine electronic and traditional voting system, there must a secure and reliable method to aggregate all votes and calculate correct results.

The way in which voters are guided through the process of electronic voting should be designed to discourage their voting precipitately or without reflection. Voters must be able to alter their choices at any point during an electronic voting process before casting their vote. They must also be able to stop the voting process without the system recording their previous choices or making them available to any other person under any circumstances. The electronic voting system must not enable any manipulative influence to be exercised over the voter during the voting process, must provide the voter with a means of participating in the election without exercising a preference (for example, by casting a blank ballot), must indicate clearly to the voter when the voting procedure has been completed, and must preserve voter anonymity.

There must be no legal impediments to interested parties who want to study the E2E-VIV system. In particular, no nondisclosure agreement or contract of any kind may be required for download and study of, or for building, testing and publishing test results for, the E2E-VIV system.

4.2.4 Assurance

There are several assurance requirements related to the implementation, documentation, and licensing of E2E-VIV systems. First, client side software—that is, any software that is expected to be used on a system serving as a voting terminal, whether a supervised machine at a polling place or an unsupervised machine belonging to a voter—must be free of known bugs on a wide range of platform and software stack combinations. The system must exhibit strong security with respect to voter authentication, such that there is no way to automate forging or invalidation of voter authentication credentials without compromising the cryptographic protocols or secrets used in the system.

All aspects of the design, architecture, algorithms and documentation for the entire Internet voting system (not just the E2E-V core) should be published and available for free download by anyone. As the system changes, all associated public documentation must be kept up to date, and no new version of an E2E-VIV system should be certified until it has up-to-date documentation.

The source code, build scripts, issue tracking system, security features, and related development information for the entire Internet voting system—all versions, for all supported platforms—should be made publicly available for free download and inspection, under a license that permits anyone to download, build, instrument, and test the system.

4.2.5 Maintenance and Evolvability

Maintenance and evolvability requirements are closely related. Election officials, or any entity engaged by election officials for this purpose, must have the right and the ability to update the election system to conform to changes in applicable law, available technology, or threats to system integrity independent of the original vendors of the system.

Election officials must also have the right and ability to patch election systems to correct flaws discovered in the algorithms, implementation, or deployment, subject to the documentation update requirement described above.

Chapter 5

Cryptographic Foundations

No existing E2E-VIV system or E2E-V protocol fulfills the requirements set forth in this report. Therefore, we cannot provide a full cryptographic system or protocol specification. The development and verification of such a specification would be one of the primary deliverables of a phase two of this project.

In order to frame that speculative future research, a formalized ideal functionality for an E2E-V system is a useful foundation for examining and comparing E2E-V protocols. This chapter provides such a foundation.

This chapter also discusses the set of technologies that should be used to mechanize and verify E2E-V protocols and their cryptographic algorithms. Mechanization and formal verification is mandatory for any E2E-V protocol that will be used in public elections.

5.1 Cryptographic Foundations

Cryptographic specifications are typically written "on paper" in peer-reviewed articles. With increasing frequency, algorithms and protocols are mechanized, either within general-purpose logic frameworks such as Coq [187] or in specialized environments such as EasyCrypt [71]. The foundations discussed in the following sections have not yet been mechanized, but must be as a first step toward any future E2E-V protocol development. A discussion of the best practices for such mechanization and verification is included in Section 5.4.

5.1.1 Ideal Functionality of an E2E-V System

In this section we introduce a template for expressing the core of an E2E-V system as an ideal functionality. An ideal functionality is an abstraction that expresses the I/O interfaces of the system with the parties that are involved in the e-voting process as well as the way the system is supposed to react to inputs. The ideal functionality specification also includes an I/O interface with the adversary, which expresses precisely the type of information that is leaked to the adversary (the output channel of the interface) and the level of influence the adversary may have on the actions taken by the functionality (the input channel of the interface).

The ideal functionality is supposed to operate in an *ideal world*, where parties have direct access to its interfaces. This means that the adversary is not able to block or manipulate the communications between other parties and the ideal functionality. The only way for the adversary to interfere is through its own interface. This emphasizes that the ideal functionality has precedence over the adversary in the ideal world. In the real world, such an ideal functionality does not exist; parties have to resort to the execution of a protocol that intends to implement the ideal functionality in reality. The conditions under which a protocol can be said to realize an ideal functionality are explained below.

The intent of the ideal functionality is to express, succinctly and in tandem, all the required properties of a system. A protocol is said to realize an ideal functionality if it is possible to translate any attack in the real world to an attack in the ideal world in a way such that, no matter how the system is operated by the parties, it is impossible to achieve any distinguishing effect between the two worlds. This indistinguishability property is the hallmark of a safe implementation of the ideal functionality. Establishing it requires a *security proof* that is constructive and algorithmic in nature. Given any real world adversary, an ideal world adversary (usually referred to as the *simulator*) is constructed that achieves the above indistinguishability property.

An ideal functionality operates in the context of an ideal world execution, a simulation that involves the following parties: the functionality itself, the environment and the adversary. The environment is the main driver of the execution that describes the sequence of actions that take place in the interfaces of the ideal functionality. It is helpful to think of these actions as serialized; however, formulations of concurrent executions are also possible. The environment is not concerned with how the ideal functionality operates, as it only provides input and receives output from the interfaces of the functionality. At the same time, the environment communicates with the adversary in some arbitrary unrestricted fashion; no assumptions are made about the interface between the environment and the ideal functionality. This unrestricted communcation with the environment is essential to ensure that the election system may be arbitrarily composed within larger systems that involve other components. If the interface between environment and adversary is specified and restricted in some way, it typically becomes simpler to realize the functionality; however, this may sacrifice the composability of the protocol.

We provide only a template for an ideal functionality for E2E-V. This emphasizes the fact that further research will be required to establish a precise formulation of this ideal functionality. Furthermore there could be many different versions of the ideal functionality capturing similar but potentially different facets of the e-voting design problem.

Interfaces of the Ideal Functionality

The interfaces of the E2E-V ideal functionality are described below. These are given to parties that the ideal functionality is able to identify. Some interfaces may be given to any party, without the functionality being concerned about the identity of that party.

The *administrator interface* is provided to the administrator of the election system. It enables the administrator to set up an election, an action that involves the description of the ballot and any restrictions and constraints that need to be applied to the ballot casting phase (e.g., how many and what choices are valid per question). The initialization message should specify the type of election function (e.g., a plurality vote) that should be applied to the inputs collected from the voters as well as the list of eligible voters, which should be a subset of the parties that the functionality is able to identify. The administrator is also responsible for opening and closing the polls.

The *voter interface* is provided to the voters. It allows a voter to cast a vote for the election that is controlled by the ideal functionality as long as the election is open. The interface is also able to return some feedback to the voter, which will enable the voter to verify that her vote has been correctly recorded and included in the final tally.

The *auditor interface* is provided to any interested party. It allows auditing of the election result and of the election process in general. The auditor interface makes public all information about the election, including the list of eligible voters. Furthermore, after the closing of the polls, this interface allows voters to verify that their votes were correctly recorded and included in the final tally.

The *adversary interface* enables the adversary to learn and influence how the ideal functionality operates in multiple ways. It enables the adversary to corrupt any party that is involved in the process. Corrupted parties are assumed to be under the control of the adversary. The identities of corrupted parties are kept by the ideal functionality, which may modify its operation depending on which parties are corrupted. Most importantly, in case of a corrupted administrator, the ideal functionality will allow the modification of voters' submitted votes.

Ideal Functionality

The following is a general template describing how the functionality reacts when receiving input from any of its available interfaces. This should be interpreted as a general guide for expressing the syntax and properties of an ideal functionality for an E2E-V system, rather than as a definitive final formulation. Using this template as a basis, several different functionalities may be derived that share the same interfaces but differ slightly in the way they operate when receiving inputs.

The functionality recognizes multiple parties, some of which are given the special role of administrator. Furthermore, it is parameterized by a predicate $P(.,.)$ that determines the *precision* of the functionality, i.e., how sensitive it is to adversarial modifications in the final tally compared to the tally calculated based on the recorded votes. Intuitively, the $P(.,.)$ predicate captures the fact that we may implement an E2E-V procedure via a protocol that cannot prevent, with overwhelming probability, an adversary from switching a handful of votes. Note that absolute precision can be achieved by setting $P(.,.)$ to be the equality predicate; in any case we require that, for any x and y, if $x = y$ then $P(x, y)$ holds.

These are the actions taken by the functionality.

- Given an "initialization" message in the administrator interface, the functionality extracts the list of eligible voters and the description of the ballot. It forwards the initialization message to the adversary. Assuming the adversary enables it to, the functionality verifies that the list of eligible voters is a subset of the list of parties it can identify and responds with success to the calling party. Otherwise, it responds with failure.
- Given an "open polls" message in the administrator interface, the functionality switches its internal state to accepting votes after receiving permission from the adversary.
- Given a "cast vote" message in the voter interface, the functionality extracts choices for the election's questions. Then, assuming that the administrator is not corrupted, the functionality notifies the adversary about the cast vote without communicating the choices of the voter; if the administrator is corrupted, the functionality also communicates the choices of the voter to the adversary. When activated, the adversary decides whether the functionality may respond to the voter. Given permission, the functionality checks that the voter is among the eligible voters and that the vote is valid given the election definition. If this is the case, it stores a record with the voter's identity and the choices she selected. It then requests the feedback for the voter to be specified by the adversary. This feedback is returned to the voter to signify that the ballot-casting submission has been accepted. The adversary is free to provide the feedback string, but the functionality restricts it to be unique: the adversary is not allowed to use the same feedback string twice. In every other respect, the precise structure of this string is left entirely to the discretion of the adversary. The feedback string is appended to the voter record and kept in the local state of the functionality.
- Given a "close polls" message in the administrator interface, the functionality switches its internal state to not accept votes anymore after receiving permission from the adversary. It then calculates the final result based on the records of the eligible voters who have voted. It forwards this tally, called the *calculated tally*, to the adversary. The adversary responds with a possibly modified tally, called the *final tally*. If the administrator is honest, the functionality forces the final tally to be equal to the calculated tally regardless of what the adversary responds. Both the calculated tally and the final tally are recorded in the local state of the functionality.
- Given a "read tally" message in any interface, the functionality requests permission from the adversary. If it receives permission, the functionality returns the final tally (this may be different from the calculated tally, if the administrator is corrupted). The functionality may also reveal, together with the final tally, the list of feedback information given to each eligible voter in association with the voter identities.
- Given a "modify voter record" message in the adversary interface, the functionality extracts the voter's choice. Then, assuming the functionality has been notified earlier that the administrator is corrupted, that the polls are still open and that the alternate voter's choice is valid given the election's definition, it changes the voter's choice in the table where it keeps the voter records, making a note that the voter's choice has been changed maliciously.

- Given a "verify election" record in the auditor interface, the functionality extracts a voter feedback string. It forwards this message to the adversary. If it receives permission, it attempts to identify the record of a voter who was given that string and return to the auditor a single bit signifying whether or not the original voter intent has been tampered with. Furthermore, it applies the predicate $P(.,.)$ to the calculated tally and the final tally and returns the output to the requestor.

Security Characteristics

Using the template ideal functionality of the above section, we describe how several required properties are captured. We examine the properties individually.

Voter Privacy. The ideal functionality ensures voter privacy up to a certain level by not disclosing the voter's choice provided to the functionality in the voter interface if the administrator is honest. We stress that voter privacy is not absolute, since the revelation of the calculated tally to the adversarial interface when the polls close reveals some information about the choices of the voters. In extreme scenarios (e.g., only a single voter casts a ballot, only a single honest voter exists among adversarial voters, or everyone votes in the same way) no voter privacy remains after the tally is revealed to the adversary. Naturally, no e-voting implementation can be expected to protect privacy in these scenarios.

End-to-End Verifiability. The verifiability provided by the system is captured by the actions taken in the auditor interface. The auditor can use the feedback provided by the system after ballot casting to check whether the voter record has been modified by the adversary. Observe that such a modification can happen only if the administrator is corrupted. It is easy to see that, if all voters audit their ballot in the auditor interface (or delegate this task to a third party that performs it), it is guaranteed that the $P(.,.)$ predicate holds between the calculated tally and the final tally. In the other extreme, if nobody audits it is easy to see that no guarantee whatsoever is given for the final tally, which may deviate arbitrarily from the calculated tally. The intermediate setting, where voters perform auditing with some probability, ensures the correctness of the tally in a statistical sense. This is one of the differences between this ideal functionality and an ideal functionality for "secure multiparty computation", a standard cryptographic notion where the output, if produced, is guaranteed to be correct independently of the actions of the adversary.

Receipt Freedom. Receipt freedom ensures that the voter does not obtain any feedback from the system that can be used to identify how she voted. Specifically, observe that, if the administrator is honest, the feedback string is generated by the adversary and is independent of voter choice. It follows that this feedback cannot carry any voter choice information. On the other hand, if the administrator is corrupted this property is not preserved: the feedback may be chosen as a function of the voter's choices and thereby violate receipt freedom. If a voter is corrupted after her vote is cast, the template ideal functionality will not divulge the voter record that it has stored in its local state. This points to the fact that any implementation of the ballot casting protocol should not leave traces from the ballot casting stage that unequivocally bind the procedure to a specific voter choice.

5.1.2 Corruption Robustness

The ideal functionality enables the corruption of any party in the election process. However, it keeps track of the parties that are corrupted and provides some security guarantees even when they are corrupted. We examine the effects of corruption from different perspectives.

Corrupt Election Administrator

A corrupt election administrator has a dramatic effect on how the ideal functionality operates. In the template ideal functionality, we express the administrator of the election as a single entity. In a protocol implementation that realizes the ideal functionality, the administrator may be implemented by a quorum of parties/trustees who collectively share the responsibility of the election administration. In this case, corrupting the administrator would amount to corrupting a sufficiently large number of trustees (the exact number is determined by the specifics of the implementation). When operating the ideal functionality in the presence of a malicious administrator, we observe that privacy and receipt freedom are lost. Moreover, voter intent as captured by the functionality can be modified by the adversary. Despite this, the functionality still offers a faithful auditing step and informs the auditor regarding the state of the voter intent, as long as the auditor provides the feedback that was obtained by the voter at the completion of ballot casting. The functionality enforces that the feedback provided uniquely identifies each voter, so any auditor that has proper feedback from a certain number of honest voters is guaranteed to be able to check the status of an equal number of voter records as preserved in the local state of the ideal functionality.

Corrupt Voters

A corrupt voter is a voter that is controlled by the adversary. Voters may be corrupted at any moment, including at the onset of the system operation and during the ballot casting process. It is expected that an adversary controlling a certain number of voters is able to shift an equal amount of votes in the final tally. However, the power of such an adversary is restricted to shifting those votes, and corrupt voters by themselves are incapable of disrupting the auditing process or jeopardizing the privacy of the other voters.

Corrupt Implementations

A corrupt implementation in the context of an ideal functionality is any implementation that fails to realize the ideal functionality. If an implementation is corrupt, this can be demonstrated via the existence of a real world adversary such that, no matter how it is transformed into the ideal world, there is an environment that produces a sequence of actions that lead to a distinguishing event between the real and the ideal worlds.

5.1.3 Absent Security Properties

The template ideal functionality does not explicitly deal with certain security aspects. The fact that they are not directly addressed is intentional, as their inclusion would make the security specification substantially more complex. In this way, the template functionality provides a baseline for security that is a minimum threshold for end-to-end verifiability and privacy. If a scheme attains at least the level of security suggested by the ideal functionality, it should then be analyzed with respect to these additional properties; whether a scheme is suitable for deployment in a certain context may depend on whether it exhibits these additional security characteristics.

Denial of service attacks. The functionality enables the adversary to prevent voters from completing the ballot-casting protocol, and also to prevent the tally from becoming available. From a definitional point of view, expressing this level of security is feasible by assuming certain qualities of the underlying communication and message passing mechanisms employed in the implementation. One way to extend the functionality to capture such details is to oblige the adversary to deliver its messages by certain deadlines; in this case, the functionality may go ahead and deliver messages via its output interfaces without requiring the adversary to enable such messages. In order to formally extend the functionality in this way, a notion of time must be introduced in the model. This may be achieved by means of a global clock functionality, relative to which deadlines can be expressed.

Coercion. Even though the functionality does not permit coercion via the voter feedback it provides, the adversary may still achieve coercion by corrupting a voter (e.g., hacking into the voter's PC) and assuming her role. Extending the model to handle such coercion is feasible by more strictly mediating the way voter corruption takes place.

Sybil attacks. The set of voters is predetermined and integrated into the functionality. Hence, the adversary cannot manipulate the list of voters. It follows that the functionality is applicable to the setting where the list of voters is predetermined and assumed to be public, and the adversary can not tamper with it. The setting where the adversary can manipulate the system via the introduction of fake identities in the eligible voter list is not explicitly addressed. Nevertheless, the functionality can produce the list of identities that have participated in voting; therefore, it is possible for auditors to verify whether the active voters correspond to real persons (e.g., by selecting a small random sample of them and performing in-person interviews).

Privacy beyond the voters' choices. The functionality leaks some specific aspects of voter behavior to the adversary, including the time that each voter casts their ballot and the final list of voters who have participated in the election. As mentioned above, this is a security feature; it enables the auditors to identify the list of active voters and validate their participation. Increasing privacy and reducing auditability by hiding the list of active voters is possible in the model.

Accountability. The template functionality does not provide any mechanism to resolve disputes between a set of voters who claim their voter records have been compromised and an election administrator who claims that set of voters intends to disrupt the election by making false claims. Such dispute resolution mechanisms may be implemented judicially outside the security model, taking into account threshold conditions; for example, if the number of complaints is below some threshold then disputes are resolved in favor of the election administrator (note that this will affect the verifiability of the election in a statistical sense). However, it is also possible to enhance the model with accountability by having the functionality present some information about the corruption state of parties. Currently, no such information is revealed to the auditor.

5.2 Contextual Analyses of Primary E2E-V Protocols

In this section we analyze a list of candidate e-voting systems from the perspective of privacy and verifiability in the context of the template ideal functionality. Further investigations will be able to determine whether the schemes below (or close variants) are capable of realizing some instantiation of the template ideal functionality. A common characteristic in all the systems we list below is the existence and general availability of a public bulletin board that unambiguously maintains the transcript of the election. This transcript is used by the auditors, together with feedback information collected from the voters, to validate the election result. These are, in some sense, minimal assumptions. If the voters are unable to have a unique view of the election transcript, it is impossible to have verifiability overall. The template ideal functionality would immediately disqualify a system where an adversary can disrupt the voters' unique view of the election transcript.

5.2.1 Demos

Demos is a system that was recently proposed by Delis et al. [65]. The central construction idea views the administrator as operating in a three move protocol known as a Sigma protocol. At the initial stage, the administrator precomputes a sequence of encodings that contain all possible ways that a voter can vote in an election and posts them on a public bulletin board. The administrator then distributes ballots that enable voters to cast their votes by essentially pointing to specific encoded information on the bulletin board. The election terminates by having the administrator post a proof to the bulletin board showing that the tally computation is correct.

A key feature of the system is that the administrator precomputes for each voter two equivalent ways to cast a ballot, and the voter is free to choose one of the two (say, "A" or "B") at random. While the A/B decision does not affect how the voter interacts with the system, it introduces voter-side entropy that is independent of the voter's PC or device and can be collected and used to ensure the calculation of the tally is correct. Taking advantage of this feature, DEMOS is capable of producing a proof that unequivocally ensures the result is correct, without having to rely on any additional assumption beyond the availability of the election transcript and the feedback from the voters.

From the implementation point of view, the voter interface requires the ballot encoding information. Given this information, ballot casting is straightforward as it only requires submitting the corresponding pointer to the bulletin board. Note that the pointer itself does not violate the privacy of the voter, since it merely points to an ciphertext that hides the actual choice under a cryptographic computational assumption (specifically, a variant of the Decisional Diffie Hellman assumption).

The verifiability of the system hinges on the fact that the election administrator will have to prove that the tally is correct by opening any of the unused ballot encodings on the bulletin board. In order to attack verifiability effectively, the administrator will have to guess the choices made by the voters in the A/B decision. While this may be feasible for just a handful of voters, Delis et al. show that the probability of not getting caught drops exponentially in the distance of the deviation of the final tally from the calculated tally (in our present terminology).

Demos (or a Demos variant) has the potential to realize some useful instantiation of the template ideal functionality given the proof arguments presented by Delis et al., assuming there is sufficient entropy in the A/B decision performed by the voters.

5.2.2 Helios

Helios [8] culminates a long sequence of previous works that used client side encryption combined with a ciphertext processing stage that breaks the connection between the identity of the voter and her choices during ballot casting. The voter casts a ballot by using a public-key encryption operation under a key that is provided by the election administrator. This key is typically generated by a quorum of trustees, who collectively control the decryption operation under a certain threshold condition.

The verifiability features of such a system are based on two mechanisms. First, the voter is allowed to challenge her client side encryption device (a PC, smartphone or other system). This is performed when the device has produced a ciphertext; the voter is asked whether to cast it or audit it. In case of an audit, the choices used for producing the ciphertext are presented and the voter obtains a transcript of information that can be used to verify that the ciphertext is properly constructed. Note that such information cannot be verified visually or by hand, and the voter is required to keep this information and verify it using a second device; in order for the check to be meaningful, this auditing device should run code that is independent from the voter's main device. After the voter has performed some audits and kept the results she can cast her encrypted ballot, which is posted to the public bulletin board. With each ciphertext that is produced, "smart ballot tracker" information— a hash of the ciphertext—is provided. This can be used by the voter to verify that the ciphertext that was generated is the same one that appears on the bulletin board.

Helios utilizes non-interactive zero-knowledge (NIZK) proofs for (i) ensuring that the encrypted votes are of the proper form, and (ii) ensuring that the decryption is performed properly by the trustees. The presence and verification of those NIZKs is essential for the verifiability and privacy aspects of the system.

Overall, auditing in Helios requires (i) the verification of the artifacts of the cast-or-audit process, (ii) checking that the smart ballot tracker corresponds to the hash of the ciphertext that is posted to the bulletin board, and (iii) verifying the zero-knowledge proofs that ensure the encrypted ballots are properly encoded and the ciphertext processing is properly executed.

The current implementation of Helios utilizes encryption with an additive homomorphic property, so that vote tallying can be executed over the submitted ciphertexts. An alternative implementation that utilizes mix-nets for ciphertext processing has been proposed [194].

Helios (or a Helios variant) has the potential to realize some useful instantiation of the template ideal functionality; it can satisfy end-to-end verifiability assuming the voters perform the audit-or-check procedure following a probabilistic strategy. Note that that such a proof will require the validity of the NIZK components, which in the present system formulation relies on an abstraction called the random oracle model.

5.2.3 Norwegian System

The Norwegian system [88] is based on a protocol developed by Scytl. The administrator generates for each voter a list of receipt codes, one for each possible choice. These are communicated to the voters ahead of time. The voters submit their encrypted votes to the administrator. Using a special ciphertext processing operation, it is possible for the administrator to extract from each ciphertext the code that corresponds to the choice of the voter and submit it back to the voter via an independent channel (e.g., if voting takes place using a PC the code can be transmitted back to a smartphone). This arguably assures the voter that the system has correctly recorded her intent without violating her privacy. The reason is that the code itself is pseudorandomly dependent on the choice of the voter and independently selected for each voter. Thus, the code by itself reveals no useful information about the choice of the voter. At the same time, the ciphertext processing step ensures that the server calculating the code has to apply the proper function and extract the proper value, and it is hard for the server to guess an alternative value given the pseudorandomness of the codes.

When the voter receives the code she can compare it with the information that is available to her and verify that the system correctly recoded her intention.

Contrary to Helios or Demos, verifiability in this system is only guaranteed if the adversary does not control both the servers and the device used by the voter to cast the ballot. If the adversary does control both, it can deceive the voter into thinking that the proper choice has been recorded when something else has been recorded instead. It follows that the verifiability guarantee for the Norwegian system is weaker than for Demos or Helios. Note that the template functionality can accommodate such weaker guarantees by suitably restricting the number and type of corrupt parties in the real execution; nevertheless, it would be preferable to avoid such restrictions if possible.

5.2.4 Remotegrity

Remotegrity [210] is a front-end system for Internet voting that can be combined with a back-end system (such as Scantegrity-II [46]) to offer a complete e-voting system with end-to-end verifiability guarantees.

During initialization, the system administrator precomputes many encoded ballots and commits to a properly formed set of tables of commitments in a public bulletin board. For each candidate choice, a unique vote code is assigned to each voter; the tables commit to a correspondence between vote codes and candidates. During the cast protocol the voter can select to audit or cast a vote. Audited ballots are spoiled, and their correspondence in the tables is revealed at the end of the election process. Cast ballots, on the other hand, record a certain vote code and the corresponding election choice is later revealed in a way that keeps the correspondence to the voter hidden.

This is achieved via the interaction of two tables of commitments that become partially open, depending on a source of public and verifiable randomness.

Assuming that the adversary is incapable of biasing the independent random source that provides the challenge for the partial opening of the commitment tables, it is possible to argue that the system satisfies a level of verifiability without sacrificing privacy.

Based on this, the system has the potential for privacy and verifiability only in case of an adversarial environment where the randomness used to challenge the administrator remains unbiased and outside the control of the adversary. It is easy to see that, if the adversary completely controls the randomness used in the challenge stage, it can produce any tally regardless of voter intent.

Several natural or human public phenomena have been proposed for providing randomness in this context; for example, stock market end-of-day quotes and weather measurements are potential candidates. However, using such sources of randomness is not straightforward because the randomness quality is limited and uniform randomness must be extracted properly.

5.2.5 RIES

The Rijnland Internet Election System (RIES) [94] was used from 2004 to 2006 for elections in Dutch District Water Boards, and in 2006 for Dutch expatriates in the parliamentary elections. RIES is simpler than the systems examined above.

The administrator produces a public-key encryption pair and one symmetric encryption key for each voter. Subsequently, voters can submit vote codes that are calculated as functions of their symmetric encryption key. The administrator, who knows all symmetric encryption keys, publishes a table of hashes on the vote codes used by the voters. A voter interacts with her device, providing the symmetric encryption key. The device prepares the vote code (which is computable using the encryption key) and maintains that vote code as a receipt for the voter. Meanwhile, the administrator decrypts all votes and posts them to a public bulletin board in some predetermined order that also depends on the symmetric encryption key.

It is possible to calculate the final tally using the information that posted to the bulletin board, by matching it with the hashed vote codes. However, the system produces receipts that unequivocally identify voter intent during ballot casting. Indeed, the system is not receipt free and therefore cannot be used as a reasonable implementation of the ideal functionality.

5.2.6 Wombat

The Wombat system [23] falls in the same category as Helios and Zeus, with the exception that it uses an on-site ballot casting setup that also preserves the original voter intent on paper. The voters generate ciphertexts with the aid of a device with a printer, which presents the voter choice in both ciphertext and plaintext forms. The voter may choose to audit the combination using the same technique as the Helios audit-or-cast mechanism. When the voter decides to cast the ballot, she tears the plaintext part and places it in a ballot box. The ciphertext part is scanned and posted to a public bulletin board. The voter retains the encoding of the ciphertext that was cast as a fingerprint to match it with the information on the bulletin board.

In order to preserve the privacy of the voters, a mix-net final step is employed to ensure that ciphertexts become disassociated from the voters who cast them. Overall the same verifiability and privacy arguments made in the case of Helios apply here (taking into account the fact that Wombat is an on-site system).

5.2.7 vVote/Prêt à Voter

The vVote system is an adaptation of the Prêt à Voter system for the Australian state of Victoria elections [61]. It retains the general structure of Prêt à Voter [171] while modifying the way the encryption of voter choices is performed to suit the style of elections in Victoria.

Prêt à Voter paper ballots are divided into a left- and right-hand side. Each ballot contains a permutation of the choices that are available to the voter. The voter is invited to enter her choices on the right-hand side while the names of the candidates appear (randomly permuted) on the left-hand side. After marking, the left-hand side is destroyed while the right-hand side is scanned; it is posted on a public bulletin board and kept as a receipt. The right-hand side also contains an encryption of the permutation used in the ballot.

Verifiability can be based on an audit-or-cast process on the printed ballot and verification of the process on the bulletin board. Specifically, the voter can choose to spoil a ballot and reveal the encryption of the permutation, which will enable the comparison of the encrypted permutation to the one physically shown on the the ballot. The voter can also track her ballot on the bulletin board, since the right hand side contains an encoding of the ciphertext. Finally, the result is revealed after a mix-net operation that includes zero-knowledge proofs to ensure that the mixing process is done correctly (namely, no ballots are substituted or removed at each mixing step).

In principle, the logic under which verifiability can be argued for vVote/Prêt à Voter is similar to that for Helios and Wombat.

5.3 Realizing Ideal Functionality

In this section we describe some general directions and tools regarding realization of the ideal functionality. We start by introducing relevant cryptographic tools and continue to describe how it is possible to mechanize the security proof of realizing the ideal functionality. We also present a case study for one system, and make the case for open protocols and software independence.

5.3.1 Commonly Used Cryptographic Tools

Several cryptographic tools play an important role in the design of e-voting schemes. We present some of these tools, which have been used in the designs of the systems discussed in the previous section.

Commitment. A *(non-interactive) commitment* scheme is comprised of two algorithms (Commit, Verify), and enables a party to produce a pair $(\psi, \rho) \leftarrow \text{Commit}(m)$ so that ψ "commits" to the value m without revealing much information about m. Specifically, a party can commit to m by sending ψ to a public bulletin board. At a later time, the party can reveal m, ρ and any other interested party can verify that the process was done correctly by running the algorithm $\text{Verify}(\rho, m, \psi)$. The two algorithms may be parameterized by a public parameter p that is available to all parties. In this case, another algorithm that generates the parameter p may be added to the description. Depending on the type of commitment, this algorithm may be required to be executed honestly. The security model requires that commitments are *hiding*, i.e., ψ does not reveal any information about m, and *binding*, i.e., it is infeasible for the party that makes a commitment to m to equivocate it and reveal a different value m'.

Public-key encryption. A *public-key encryption* is comprised of three algorithms (Gen, Enc, Dec). The algorithm Gen produces a pair of public and secret key, denoted by (pk, sk). Algorithm $\text{Enc}(pk, \cdot)$ applied to input plaintext m returns a ciphertext ψ that corresponds to m. Algorithm $\text{Dec}(sk, \psi)$ returns the plaintext that corresponds to the input ψ. The security model requires at minimum that an adversary, given the public-key pk, is incapable of distinguishing between two ciphertexts that correspond to different plaintexts even if such plaintexts are adversarially chosen.

Additively homomorphic encryption. A *homomorphic (public-key) encryption* scheme adds the following property to the $\text{Enc}(pk, \cdot)$ algorithm. First, the space of plaintexts is given a group structure over a binary addition operation $+$. Then, given $\psi_1 = \text{Enc}(pk, m_1)$ and $\psi_2 = \text{Enc}(pk, m_2)$, it is possible to compute a ciphertext ψ that corresponds to an encryption $\text{Enc}(pk, m_1 + m_2)$. Furthermore, we also require that if at least one of ψ_1, ψ_2 is sampled uniformly over all ciphertexts, the output ψ follows the uniform distribution over all ciphertexts of $m_1 + m_2$.

Additive homomorphic encryption enables processing of ciphertexts in various ways that are useful in e-voting systems. Two important uses of such schemes are as follows: (i) Given k ciphertexts $\psi_1, \ldots, \psi_k$ encoding $v_1, \ldots, v_k \in \{0, 1\}$, it is possible to derive a single ciphertext ψ that encodes $T = \sum_{i=1}^{k} v_i$. Such a T can be used to derive the tally of an election if each plaintext v_i corresponds to an election choice made by the i-th voter. (ii) Given $\psi = \text{Enc}(pk, m)$ it is possible to "refresh" the randomness of ψ by processing ψ together with $\text{Enc}(pk, 0)$. The resulting ciphertext is uniformly distributed over the ciphertexts encoding m.

Secret-sharing. A *secret-sharing* scheme is comprised of two algorithms (Gen, Rec) that operate as follows. Given parameters (n, t) and a value s, the algorithm Gen produces n "shares" $s_1, \ldots, s_n$. The algorithm Rec, given any subset of size at least t from the set $\{s_1, \ldots, s_n\}$, reconstructs the value s. The security guarantee that we require for a secret-sharing scheme is that, if the adversary has any set of size less than t values from $s_1, \ldots, s_n$, it should be incapable of finding any non-trivial information about s. The value t is called the *threshold* of the secret-sharing scheme.

An additional property for secret-sharing schemes is verifiability, where the reconstruction algorithm is capable of detecting shares that are incorrect based on some public information provided by the Gen algorithm.

Threshold encryption. A *threshold (public-key) encryption* is equipped with a multiparty protocol that implements the Gen() public/secret key generation algorithm among a set of parties. Specifically, this protocol is parameterized by two integers (t, n) and enables a set of n parties (sometimes called trustees) to produce the value pk as well as a secret-sharing of the value sk with threshold t. The shares of the secret key sk are kept by the trustees and can be revealed when it is time to decrypt a ciphertext.

A threshold encryption scheme is also capable of achieving threshold decryption if there is a protocol that implements the algorithm Dec() so that each trustee uses her own share of the secret-key sk but is not required to reveal it. Threshold decryption is particularly important in practice, since it is typically desirable to apply the decryption function Dec selectively only to specific ciphertexts that may be identified by the system that utilizes threshold encryption.

The property required by a threshold encryption is similar to that of secret-sharing. The adversary should be incapable of finding any information about a target ciphertext if it controls less than t trustees. Importantly, the adversary should be incapable of doing so despite the fact that it may be participating in various threshold decryption operations running concurrently on ciphertexts other than the target ciphertext.

Zero-knowledge proofs. A *zero-knowledge (ZK) proof* is a protocol between two parties, called the prover and the verifier, that is associated with a language $L = \{x \mid \exists w : R(x, w)\}$, where R is a polynomial-time predicate in a parameter k and x and w are strings of length k. The protocol enables the prover to convince the verifier that she is in possession of a "witness" w regarding the fact that $x \in L$.

The properties required by a ZK proof are completeness, soundness and zero-knowledge. Informally, completeness requires that the verifier accepts an honest prover, soundness expresses that the prover cannot cheat (e.g., that she cannot convince the verifier if she does not know the witness) and zero-knowledge that the verifier cannot learn anything significant about w from interacting with the prover beyond the fact that $x \in L$. Sometimes it may be the case that $x \in L$ is a given fact and the protocol has the objective to demonstrate that the prover is in possession of the witness w. Protocols with this property are called ZK proofs of knowledge.

An important variation of ZK proofs are non-interactive ZK (NIZK) proofs, where the prover is capable of producing a string π in one move that by itself can convince the verifier regarding the status of the statement $x \in L$ (or that the prover is in possession of the witness w). A NIZK requires a public parameter p that should be honestly produced (independently of the prover and the verifier).

Mix-nets. A *mix-net* is a protocol executed sequentially by a set of servers on a set of ciphertexts that originate from a public-key encryption scheme. Specifically, a mix-net can be built upon a single algorithm Shuffle() that is given the public-key pk as well as a sequence of ciphertexts $\vec{\psi} = (\psi_1, \ldots, \psi_n)$. The algorithm produces a sequence of ciphertexts $\vec{\psi}' = (\psi'_1, \ldots, \psi'_n)$ as well as a "proof" π. The proof π can be in the form of a NIZK that ensures the following fact about the values $(pk, \vec{\psi}, \vec{\psi}')$. Let M be the sequence of plaintexts that in the i-th position is equal to $\text{Dec}(\psi_i)\rangle$, and similarly M' the sequence of plaintexts that in the i-th position is equal to $\text{Dec}(\psi'_i)\rangle$. It should hold that there is a permutation μ over n elements such that the j-th element of M is equal to the $\mu(j)$ element of M' for all $j = 1, \ldots, n$.

Given the Shuffle() algorithm, observe that we can apply it sequentially on the same sequence of ciphertexts. If a sequence of m servers apply Shuffle() one after the other, it is easy to see that the correspondence between the original sequence of ciphertexts and the final sequence of ciphertexts will be hidden provided that at least one server remains honest.

5.3.2 Other Potentially Useful Cryptographic Tools

There are several other cryptographic tools that have not yet seen much application in the context of E2E-V protocol design. Several are briefly summarized here, in part because they represent opportunities for novel research directions in E2E-V protocol design. They also described because some naive proposals for election systems include the use of these tools, and should be identified and rejected.

Hashchains and blockchains. A *hashchain* is a data store that holds a ledger of sequential records in a cryptographically secure fashion [99]. Hashchains are commonly used to record a sequence of causally dependent events with cryptographic integrity and non-repudiation. Within the context of election systems, hashchains are a common means of recording privacy-preserving election logs for post-election audit, constructing digital ballot boxes, crafting public bulletin boards that contain evidence of an election's correctness and security properties, and more.

A hashchain H is constructed by the repeated application of a cryptographic hash function h. Linear hashchains are of the form $H^k = h(x_k \otimes h(...h(x_2 \otimes h(x_1 \otimes h(\perp \otimes \perp)))))$ where each (identical) hash h is applied to a fusion $\otimes$ (often defined as xor) of the previous hashchain and a new ledger element x. The bottom element $\perp$ can be one of several potential root values (e.g., a prefix of the public key of a given election), depending upon the context of the application of the hashchain.

Non-linear hashchains are trees of linear hashchains [22, 24]. They are useful in the context of disconnected devices with causally or temporally connected ledgers (e.g., disconnected machines used in a given election).

A *blockchain* is a distributed data store that holds a ledger of transactions in a cryptographically secure fashion. It is a kind of distributed hashchain, where multiple computers compute and communicate to determine the next data element of the hashchain/tree using a consensus protocol. Blockchains often rely upon a cryptographic work factor to determine consensus and to prevent manipulation of the ledger by powerful adversaries; the most publicly visible example of a blockchain that works in this way is the Bitcoin blockchain. [146]

Proposals to use blockchains for elections are plentiful, but have been shown to be naïve in most instances and inappropriate as a foundation for a public election E2E-V protocol. The only reasonable proposal for the use of blockchains for parts of an election protocol is from Clark and Essex [50].

Multi-party computation and linear secret sharing. *Secure multi-party computation* (MPC) is a collaborative privacy-preserving computation technology. MPC permits a (typically small) collection of parties to compute a collaborative result without any party gaining any knowledge about the inputs provided by other parties, except for what can be determined from the output of the computation [41, 93].

Since the computation takes place on encrypted data, it can be performed on public systems (such as in a public cloud infrastructure). Moreover, since systems can communicate using a public protocol to compute collaboratively, parties to the computation can be implemented by multiple cooperating or competing organizations.

In the kind of MPC known as *linear (or additive) sharing*, computation proceeds on data that appears entirely random [58]. Certain operations, such as addition or logical exclusive OR, can be performed locally; however, operations such as multiplication and logical AND require network communications among the parties. Consequently, the computational overhead of MPC is large, and MPC is orders of magnitude slower than computing on unencrypted data.

However, efficiency improvements over the last few years have shifted the potential applicability of MPC from micro-benchmarks to user-level applications, including some that have a data volume comparable to elections. Consequently, there may be real opportunities to use MPC for E2E-V elections.

Functional encryption. *Functional encryption* (FE) is a kind of public-key cryptography in which possessing a secret key permits one to learn a function of what the ciphertext is encrypting, and nothing more [30].

More precisely, a FE scheme for a given functionality F consists of the following four algorithms:

1. $(pk, msk) \leftarrow Setup(1^{\lambda})$: creates a public key pk and a master secret key msk;
2. $sk \leftarrow Keygen(msk, k)$: uses the master secret key to generate a new secret key sk for value k;
3. $c \leftarrow Enc(pk, x)$: uses the public key to encrypt a message x; and
4. $F(k, x) \leftarrow Dec(sk, c)$: uses secret key sk to calculate a function of the value c encrypts.

FE's primary use is in encrypted databases whose security properties are determined, in part, by the computations permitted on their encrypted data [161]. Consequently, FE may prove useful in storing, or computing on, election data such as ballots and audit logs.

FE encryption generalizes several existing primitives, including *identity-based encryption* (IBE) and *attribute-based encryption* (ABE), both of which may be useful in the context of authentication in E2E-V protocols [95, 176].

Fully homomorphic encryption. *Fully homomorphic encryption* (FHE) is a more powerful type of homomorphic encryption than those previously mentioned. FHE permits arbitrary computation on encrypted data, but is prohibitively slow (several orders of magnitude slower than unencrypted computation) and requires enormous ciphertexts (dozens to hundreds of megabytes in size).

It is unclear whether FHE has potential application in the context of E2E-V protocols. However, given increasing interest in FHE and the availability of open source implementations of FHE libraries, it is likely to applied in the context of cloud deployments of election systems.

Verifiable computing. *Verifiable computing* is a new area of research and development that focuses on the offloading of computation to an untrusted system (for example, an untrusted e-voting machine or a cloud hosting service) while still having a means by which to check that the computation was performed correctly.

Several technologies exist to help verify that a computation performed by untrusted workers is correct, including the use of secure coprocessors, Trusted Platform Modules (TPMs), interactive proofs, and more [124, 158, 183]. These are either interactive verifications, which require the client to interact with the worker to verify the correctness proof, or are non-interactive protocols that can be proven in the random oracle model.

The utility of verifiable computing in the context of E2E-VIV is obvious, but the state of the art is still far from being able to handle the data volumes mandated by public elections. It is to be expected that new results in verifiable computing, and E2E-V protocols that presume the existence of efficient verifiable computing, will be forthcoming.

5.4 Formal Mechanization of Ideal Functionality

As mentioned above, in order to contextualize E2E-V protocol designs and especially to objectively compare and contrast them, the ideal functionality must be mechanized.

The premiere environments in which to perform such mechanization are Coq [187], EasyCrypt [71], CryptoVerif [60], and Cryptol [132]. Each has pros and cons, and it is certainly reasonable to consider mechanizing protocols and their dependent algorithms in multiple environments. In general, any framework chosen must be trusted by cryptographers, be open source, and have liberal licensing terms.

There are several other excellent useful environments that should be considered for future mechanization work, including higher-order frameworks such as PVS [177], Isabelle [112], and HOL [102], and protocol-centric tools such as F* [76], CVK [62], and ProVerif [28], which we will not describe in detail here.

Coq. Coq is a general purpose logical framework based upon the Calculus of Inductive Constructions. It has a small trusted core, proofs are first-class constructs, and several libraries are available for reasoning about not only cryptographic algorithms and protocols, but also the correctness of programs written in a variety of languages. A significant amount of recent research focusing on the verification of cryptographic algorithms and protocols has used Coq, including Princeton's work on formally verifying SHA and HMAC, Harvard's work on reasoning with the Foundational Cryptography Framework, and IMDEA's early work on their precursor to EasyCrypt, CertiCrypt.

EasyCrypt. EasyCrypt is a toolset for reasoning about relational properties of probabilistic computations with adversarial code. Its main application is the construction and verification of game-based cryptographic proofs. Initial applications of EasyCrypt focused on encryption and signature schemes, but recent extensions reason about the security of cryptographic systems that achieve specific functionalities through intricate combinations of several primitives. These developments have significantly expanded the scope of potential applications of EasyCrypt, as reflected in the recent formalization of secure function evaluation and verifiable computation. Moreover, they have enabled the formalization of examples that were previously out of scope, for instance modular proofs of security for key-exchange protocols and E2E-V protocols like Helios. EasyCrypt was custom-designed by Barthe and his colleagues at IMDEA and INRIA.

CryptoVerif. CryptoVerif is a computationally sound mechanized prover for cryptographic protocols created by Blanchet and his collaborators. It has been used to reason about the secrecy and correspondence properties of numerous protocols and gives a bound on the probability of any attack.

Cryptol. Cryptol is a domain-specific language for specifying cryptographic algorithms. A Cryptol implementation of an algorithm resembles its mathematical specification more closely than an implementation in a general purpose language. Using a specification in Cryptol, programmers can generate their own test vectors, prove theorems, and (using other tools) verify equivalence to their own programs, or even generate code or hardware from the specification.

5.4.1 Recommendations

Based upon our experience, a promising approach would be to use Coq to formalize systems and reason about implementations written in C (using CompCert and VST), Cryptol to specify cryptographic algorithms and simple protocols and reason about their correctness and their implementations' (in LLVM or JVM), EasyCrypt to reason about algorithms' and protocols' security properties, and F* to specify and reason about protocols that need a Javascript or .Net implementation.

5.5 Specification of Open Protocols

E2E-V protocols, while cryptographic protocols, are at their core just protocols. They specify the means by which different subsystems communicate with each other. If we presume that different subsystems may be implemented by different parties, and that multiple independent implementations of critical components (such as tabulation and verification subsystems) are mandatory, then the precise specification of open protocols is critical.

The best practice for protocol specification is to formalize the protocol using a precise mechanization, provide a reference implementation, and provide a means by which any implementation can be validated against the specification. Given the aforementioned toolsets—ProVerif, EasyCrypt, CryptoVerif, and F*—specifying the protocols is straightforward. Verifying the protocols' correctness and security properties is a much more challenging, though not insurmountable, proposition.

The real challenge is in verifying an implementation's correctness. Full blown formal verification of implementations, while possible, takes a significant amount of effort, typically several man-months of work. Moreover, if a specification is not written with the intention of supporting a protocol under evolution, re-verifying changes in a protocol implementation can require as much effort as the original verification.

Rigorous validation of a protocol, on the other hand, takes significantly less effort and is more flexible in the face of protocol evolution. Recent work from Cambridge focusing on the rigorous specification and implementation of TLS [138] in a pure functional style exemplifies the kind of reference implementation that should be written for an E2E-V protocol.

5.6 The Case for Software Independence

A voting system is software independent if an undetected change or error in its software cannot cause an undetected change or error in an election outcome. One way to achieve software independence is to endow the voting system with the capability to record voter intent physically in a way that is immediately verifiable by the voter at the time of ballot casting. Given such a record, one would subsequently perform a random audit to ensure that the published election outcome (generated by software) does not deviate from the result that would have been calculated if the physical record was used in the tallying process.

A notable example of such a procedure currently in place is in the state of Connecticut, where optical scan voting machines are used. The marked paper ballots are retained after the election in each precinct. After the end of the election when the tally is announced, a sample of precincts is randomly selected at the state level and a manual hand count is performed. The results of the hand count are audited and a statistical analysis is conducted to confirm the e-voting tally at a certain confidence level. Antonyan et al. [13] provide more details regarding this audit procedure.

Chapter 6

Architecture

The *architecture* of a computing system, akin to the architecture of a purely physical artifact like a bridge or a building, is its high-level structure. Just as when designing a bridge, many choices must be made when designing a computing system; these choices are driven by the system's requirements, both technical and non-functional, as well as by external factors such as the availability or affordability of computing hardware or network bandwidth. In this chapter we describe the architectural issues associated with E2E-VIV systems, present a model encompassing the various possible architectural choices for such systems, and briefly explore some architectural variants.

It is important to note that we are *not* making a concrete recommendation for a specific E2E-VIV system architecture. The cryptographic foundations of E2E-VIV protocols have been developed to a point where we have fairly high confidence in their ability to detect errors in election outcome (whether unintentional or malicious) and protect ballot privacy. However, E2E-VIV protocols do not prevent such errors, and an implemented E2E-VIV system should minimize the chances of there being such errors. For this reason, there are many open engineering issues associated with actually building, running, and maintaining an E2E-VIV system that fulfills its requirements in the face of both routine/expected failures and a wide range of security threats. Because of these open engineering issues, architectural experimentation—preferably, empirical testing of various possible architectures to determine the most appropriate one(s) to deploy in real-world election scenarios—is vital to actually implementing a successful E2E-VIV system.

6.1 Non-Functional Requirements Forcing Architectural Factors

Several non-functional requirements of E2E-VIV systems force the inclusion or consideration of specific architectural factors. We consider each of these in turn.

6.1.1 Certification

Voting systems in most jurisdictions are subject to certification requirements. The EAC and NIST have developed a set of Voluntary Voting System Guidelines (VVSG) at the federal level, and the EAC launched a Voting System Testing and Certification Program in 2007; prior to 2007, voting systems were typically tested and certified by the National Association of State Election Directors (NASED). The EAC's guidelines are voluntary; however, many jurisdictions have mandated adherence to them, so an E2E-VIV system that is to be widely deployed must follow them as well as any other state and local guidelines.

Certification processes generally require that a specific implementation of a system be presented to a certification laboratory; in the E2E-VIV context, this would likely consist of a complete set of source files, plus binaries built to run on specific platforms and with specific external dependencies, and any custom hardware to be used in the system. Certification is typically an expensive process, often costing many times more than developing the system in the first place. Moreover, certification generally applies only to the system as submitted; any changes to the system require a full or partial recertification at additional expense.

This certification regime presents a sharp contrast to today's typical software lifecycle. Most modern software systems, especially those developed using agile techniques, are updated frequently to address implementation issues and respond to changing customer needs. For example, since its first release in September 2008 the Google Chrome web browser has seen 42 major releases—an average of about one every two months—and many minor patch releases in between. That sort of release cycle is simply not an option for an E2E-VIV system, unless the voting system certification regime changes dramatically. Architectural decisions made during the system's development must therefore be undertaken with great care, because major architectural changes will be extremely expensive.

6.1.2 Abstraction

The software in an E2E-VIV system must be high assurance, and must, as discussed above, undergo a certification process before it can be used in actual elections. Thus, the software must be designed and implemented with a level of abstraction that enables the generation of convincing evidence for its correctness. This requires development techniques that emphasize formal specification and verification, which divide the software system into relatively small components with well-defined, well-constrained interfaces. Each component can then be verified individually with respect to its specification; moreover, component behaviors with respect to external communication can be formally characterized in ways that allow for verification of composed subsystems.

6.1.3 Deployment

There is a wide spectrum of possible deployment scenarios for an E2E-VIV system, each of which leads to certain decisions about its architecture. At one extreme, the servers for an E2E-VIV system could be hosted on a bespoke server cluster, built from the ground up specifically for the system and housed in a facility under the physical control of electoral authorities or their authorized representatives. At another extreme, the servers could be hosted on a commodity cloud computing infrastructure such as Amazon EC2 or Microsoft Azure where electoral authorities have no physical control over the servers. A third extreme would see the system implemented in a purely peer-to-peer fashion, with the system's functionality distributed among all participating computers and no specifically-designated servers. The choice within this spectrum has significant impact on both system availability and system security requirements.

6.1.4 Threats

Mitigating the potential threats to E2E-VIV systems also leads to various architectural choices. The following are some of the threat vectors and mitigation strategies that need to be considered.

Single Point of Failure. Any single point of failure, such as a single server that contains essential data without which the system can no longer function, is a tempting target for attack. Therefore, the architecture should attempt to minimize or eliminate such failure points.

Easy DDoS Targets. The use of fixed IP addresses (or address ranges) for E2E-VIV system components can open the system to denial of service attacks, limit deployment flexibility, and make it more difficult to recover from failures. The architecture should be chosen such that IP addresses need not be hard-coded; preferably, they should be changeable on-the-fly (i.e., during a live election scenario) if necessary to protect or restore system integrity.

DDoS Mitigation. Content distribution and network protection services such as CloudFlare [52], which route traffic for their clients through their own high-bandwidth global content distribution networks, can detect and protect against DDoS attacks. They can also provide other benefits, such as higher availability and improved response time. Such services could easily be used to deliver static data to clients in an E2E-VIV system; however, although they support technologies such as WebSockets for dynamic data, they cannot necessarily protect systems that have complex back-end architectures and client-server interactions. It is unclear whether such services can successfully be used for all interactions in E2E-VIV systems without compromising privacy or other requirements; even if it is possible, doing so would certainly require specific considerations in the design of the system architecture.

Non-Standard Foundations. One way of countering threats is to build the system using non-standard foundational techologies. For example, instead of being built atop general-purpose operating systems like Linux or NetBSD, E2E-VIV system components could be implemented as *unikernels* [136], effectively single-purpose application/operating system combinations that run directly atop hypervisors such as Xen [206]. Unikernels, written using technologies such as Mirage [144] and HaLVM [189], have considerably less code than general purpose operating systems; each one performs a specific task, and they communicate amongst themselves in the same manner as machines in a distributed system. This implementation style can improve security in general, by reducing the effort required to demonstrate the security of each component and by minimizing potential attack surfaces once the components are deployed.

Secure Multi-Party Computation (MPC). A common situation in distributed computing systems is that a multiple systems, each having its own secret information, want to collaboratively compute a result based on all their secret information without sharing the secret information itself. Secure multi-party computation (MPC) enables exactly this, letting the individual parties to a computation keep their local secrets while computing global functions. MPC functionality can be useful in distributing trust among multiple individuals or organizations in a complex system, mitigating several of possible threats; we describe the distribution of trust in more detail in the following section.

6.1.5 Distributing Trust

The distribution of trust in an E2E-VIV system is critical: if the system is not trustworthy, the election results generated by it are inherently suspect. The following are descriptions of the various aspects of the system that must be trusted, and some possible ways to establish that trust.

Trusting the Electoral Authority. In general, the electoral authority is responsible for the integrity of the election outcome, the privacy of the votes, and the availability of the election system. Most E2E-VIV systems do not require the public to trust the electoral authority or election technology (including election servers run by election officials or any others assigned to this task by the electoral authority) to detect election integrity problems. The systems are designed to enable the public to detect integrity issues without requiring trust in electoral authorities or election technology, as long as the website is secure with the ability to detect tampering. Additionally, these systems typically use threshold cryptography so that vote privacy is not compromised as long as a minimum number of election officials behave as required by the protocol. The use of threshold cryptography also influences the ability of individual election officials to affect a denial of services.

In threshold cryptography, the cryptographic keys that allow for results to be generated by the election system are divided into some number of shares, a smaller threshold number of which are required to actually generate results. For example, the keys might be divided into 10 shares, each of which is entrusted to a different official and at least 7 of which are required to generate the tally. This also means that fewer than 7 officials cannot compromise vote secrecy. This allows the election to proceed in situations where some of the election officials are unable (due to accident, illness, etc.) or unwilling (due to denial of service efforts to stall the election outcome from being tallied) to produce the keys, as well as providing a check against vote secrecy violations by requiring at least 7

of the 10 officials to collaborate in generating the tally. The check against denial of service can be strengthened even further with public ceremonies surrounding key generation, share distribution, and tally generation so that the specific electoral officials involved are publicly known and can be held accountable for efforts to stall the tally computation and its verification.

Another mechanism to increase trust in the electoral authority is access restriction; if the election system allows physical access only via computing systems at specific, publicly-known and well-secured locations, and only during specified time frames, it is far more difficult for a corrupt official (or group of officials) to manipulate the election results without being detected.

Finally, the use of an append-only public bulletin board to record encrypted ballot information in a tamper-evident fashion, combined with publication of sufficient information about the election protocol to allow members of the public to validate that their own ballots were cast as intended and counted as cast, also acts as a significant check against corruption by the electoral authority.

Trusting the Software. The software in an E2E-VIV system must be trusted to fulfill the system's requirements. Making all development artifacts freely available as open source is an important step toward such trust, as it allows for independent verification that the source code has been designed and implemented appropriately. However, the mere fact that it is open source is not enough to make a piece of software trustworthy; there must be evidence that the running software actually corresponds to the open source code, has passed all required testing, and has not been corrupted after installation.

Several mechanisms are available for increasing the trustworthiness of deployed software. One is code signing; the system vendor can digitally sign the final binary distribution of the software and make the signature public, allowing any observer with sufficient access (in practice, this would likely be the electoral authority) to validate the signature against the actual deployed binaries. Self-certification is a variant on code signing, where the software computes a signature on itself at startup time and compares it against a known value; this can detect accidental corruption (data storage errors, spurious bit flips, etc.) but generally not malicious corruption, because an attacker that successfully corrupts any part of the software could also corrupt its self-certification mechanism. Similarly, a voter relies on the client to check the signature whether using code signing or self-certification, and a malicious client can modify the software and pretend it passed the signature check.

Proof-carrying code [147] allows software to be shipped with accompanying formal correctness proofs that can be easily verified by a theorem prover or certifying compiler at runtime. Such proofs can help increase the trustworthiness of software and can easily be included in small critical sections of an E2E-VIV system, but are impractical for verifying the correctness of the entire system.

Another way of increasing trust in the software is to ensure that all implemented protocols are open, with publicly available specifications, and that multiple redundant implementations of the protocols are used throughout the system. Ideally, the redundant implementations should have different low-level designs and be implemented using different programming languages. Thus, in order to corrupt the system, it would not be sufficient to corrupt a single protocol implementation; instead, all the implementations would need to be corrupted in a consistent way that could not be detected by observing differences in their behaviors.

Trusting the Servers. In addition to trust in the software, trust in the servers on which the software runs is critical. Perhaps unintuitively, one approach is to assume that the servers can't be trusted and design all the protocols and processes in the system appropriately given that assumption. The property of *software independence*—an undetected change or error in the software must not cause an undetectable change or error in an election outcome—is an example of this approach. Even if the software is fully trusted, an untrusted server can corrupt it in arbitrary ways; software independence doesn't stop such corruption, but does guarantee that if it affects the election results, it will be detected. E2E-VIV systems are designed to be software independent.

Even though distrusting the servers is useful in ensuring robust protocol and system design, steps should still be taken to make the servers more trustworthy. One way to do this is to minimize the number of components that must be trusted; an example would be using servers that support unikernel architectures, as described in Section 6.1.4, to ensure that there is little to no other potentially vulnerable software software, including OS libraries, running alongside the E2E-VIV components.

Another way to increase trust in the servers is to deploy them in a dynamic, possibly public, cloud infrastructure. In this way, the particular server responsible for any given part of the system at any given time is unknown. Assuming that the entire cloud infrastructure is not compromised (for example, by a corrupt high-level insider), this can significantly reduce the likelihood of a successful attack against the servers.

Designing the system with a fully peer-to-peer architecture can help to increase its trustworthiness, as there will no longer be single points of failure; moreover, as shown in the seminal Byzantine Generals result [128], corrupted peers in a distributed system can always be detected as long as more than 2/3 of the peers remain uncorrupted. With appropriate use of cryptography, that fraction can be improved significantly such that, in the most extreme case, even a single uncorrupted peer can detect corruption of the entire rest of the peer network.

A more radical way of increasing trust in the server infrastructure as a whole is to pervasively use MPC, such that no single server ever needs to be completely trusted. While some servers may be corrupted, they will be detectable, and with a suitably redundant system design the non-corrupted servers will still be able to carry out the necessary computations for the election.

Trusting the Network. The Internet is inherently insecure, so measures must be taken to ensure the integrity of the system's Internet-based communications. The first and most obvious of these is that all communication among components in the system should use the TLS protocol. The certificates used to establish TLS connections should be *pinned*, meaning that when a client wants to connect to a server, the client already has information about the server's security certificate (or information about a set of certificates, if more than one is acceptable). If an unexpected certificate is provided by the server, even if the hostname on the certificate matches the hostname of the server, the connection fails.

Certificates used in E2E-VIV systems should have signature chains that involve not only a certificate authority (like Comodo, Symantec, etc.) but also the electoral authority or other appropriate government entity; requiring the electoral authority to be involved in the signature chain ensures that certificates cannot be issued for malicious purposes by a rogue certificate authority and used to compromise a running E2E-VIV system.

Finally, all communications of critical information (ballot information, voter information, logs, audit data, etc.) in the system—even over TLS-encrypted connections—should be carried out using custom cryptographic protocols, so that even in the unlikely event of a TLS compromise the data is protected.

Trusting the Voting Client. One clear "weak link" in the security of an E2E-VIV system is the voting client, which must by definition—regardless of its implementation technology—involve untrusted machines belonging to individual voters. There are two potential issues: first, the voting client software itself might be corrupted, either in transit to a voter's machine or after installation; second, a voter's computing environment might be corrupted in a way that compromises the voter's interactions with the software.

There are various ways to ensure that the voting client software remains uncorrupted. One technique for doing so with native applications is application signing, where each application binary is digitally signed before distribution and the signature is checked at runtime. Trusted Computing techniques such as remote attestation can provide even stronger guarantees, but rely on the presence of a Trusted Platform Module (TPM) in the voter's computer. However, the fact that TPMs are not universally deployed (for example, Apple has not shipped a computer, tablet, or phone with a TPM since 2006) means that E2E-VIV systems cannot rely on Trusted Computing techniques.

Implementing the voting client as a web application can alleviate some concern about the corruption of distributed native application binaries, but makes it more difficult to ensure the voting client's integrity and reason about its behavior. One problematic aspect of web applications is that JavaScript can be interpreted differently not only across platforms, but also across browsers on the same platform; there are currently four different main-

stream JavaScript engines—V8 for Google Chrome, Spidermonkey for Mozilla FireFox, Nitro for Apple Safari, and Chakra for Microsoft Internet Explorer—plus a host of others used by other browsers on various platforms. In addition, the JavaScript language itself is not particularly amenable to the types of analysis required for high assurance software. Progress has been made toward enabling high assurance JavaScript, including techniques such as Certified JavaScript [119], several JavaScript tools from the Center for Advanced Software Analysis [39], and Microsoft Research's Crypto Verification Kit [62]. However, significant research is still needed in this area.

One potential approach for implementing a high assurance voting client as a web application is to start with a high-level language that supports the sorts of analysis necessary to provide the required correctness and security guarantees. Mozilla's asm.js [19] is a strict subset of JavaScript that can be used as a target for compiling such higher-level languages. One nice side effect is that this subset of JavaScript is easy to reason about, and generally behaves identically across JavaScript runtime environments. With some research and tool development, it should be possible to develop a high assurance voting client, compile it to asm.js, and deploy it as a web application.

With respect to the second issue, corruption of the voter's computing environment, there is a wide range of possible threats. One, which we do not know how to address as part of an E2E-VIV system implementation, is surreptitious logging/monitoring software that records the voter's actions and transmits them to a third party. Detection and prevention of such monitoring can be improved through standard secure computing practices such as malware scanning and the use of reverse firewalls (preventing the computer from making outgoing connections that are not approved by the user or by system security policies). However, we can only recommend that voters take such protective measures, and the voting client software cannot detect whether they are actually in place. Additionally, it is possible that a rare effort towards such monitoring is not detected by malware scanning or the use of reverse firewalls.

Another threat is posed by malicious web browsers or OS libraries that may deceive the voter into taking unintended actions. Such malicious code might do any or all of the following: manipulate the appearance of the voting client such that a voter believes she has voted for a particular candidate or choice when she has actually voted for a different one; block or corrupt communications between the voting client and the rest of the E2E-VIV system, causing the voter to unintentionally spoil ballots or otherwise interfering with vote casting; and spoof the client-side verification screens to make a voter mistakenly believe she is voting when she is actually not interacting with the rest of E2E-VIV system at all. Many of these threats can be partially or completely mitigated through judicious use of cryptography and good user interface design—for example, prominent use of confirmation codes transmitted out-of-band, such as via postal mail, to authenticate communications between the voter and the system in both directions—but there will always be the possibility of malicious action on the part of untrusted client systems. A good E2E-VIV system will enable a voter to detect any such effort that changes election outcome. Additionally, an E2E-VIV system with dispute resolution will enable a voter to prove that there is a problem.

Trusting the Data. As previously discussed, communications can be protected using TLS and custom cryptographic protocols. However, the communicated data needs to be committed to stable storage at some point, in order to be verified, tallied, and audited. Standard database technology is inappropriate for this purpose because of the data integrity, confidentiality, and provenance requirements that must be met in an E2E-VIV system. Data must be encrypted at rest to protect against straightforward physical theft of storage devices, and should remain encrypted during as much of the system's computation as possible to protect against manipulation or disclosure by either malicious software or individuals with administrative access to the data store.

One way to prevent data manipulation, or at least make it nearly infeasible, is to effectively make critical data in the system "write once" by using a technique like hash chaining to ensure its integrity; if a piece of data is changed after being written, the hash chain for all subsequently-recorded data must also be modified in order to "hide" the change. This makes data manipulation far more difficult, though still technically possible if the system is sufficiently compromised and its usage is not closely monitored (recomputing the entire hash chain would generally require significant computational resources). Replication of the data and the hash chain in multiple locations that must be accessed independently further prevents manipulation.

One way to prevent inappropriate disclosure of data while retaining the advantages of well-known and well-understood database technology is to use a tool like CryptDB [161], which allows for storage of and secure computation with encrypted data in standard database systems like MySQL and Postgres. Other possibilities include novel MPC frameworks like ShareMonad [129] and partial or full homomorphic cryptographic schemes such as ElGamal and BGV that enable computation of encrypted results over encrypted data.

The use of a public append-only bulletin board with vote encryptions published in a tamper-evident and encouraging voters and observers to frequently check the board provides a means of detecting data manipulation, while the above safeguards make it additionally difficult.

Trusting the Voter. The voter, like the voting client, is a "weak link" in the security of an E2E-VIV system. It is possible for voters to compromise their own authentication credentials intentionally or otherwise, allowing others to vote on their behalf; to falsely challenge their own legitimately cast votes either maliciously or because of mistaken memory; and to be misled into either "voting" on a system other than the actual E2E-VIV system or missing the time window for ballot casting.

In an unsupervised system, voters are responsible for their own security. However, the vast majority of people lack the knowledge and tools necessary to keep their computers secure. The electoral authority can strongly encourage certain security measures, such as use of antivirus/antimalware software on appropriate platforms, manual checking of SSL certificates to ensure that they are signed by the electoral authority, and installation of up-to-date OS, browser, and E2E-VIV system software and security patches. However, there will inevitably be voters who do not follow these recommendations, and it is generally not possible—especially if it is web-based—for the E2E-VIV system to evaluate a voter's compliance with them and act accordingly. Moreover, even voters using completely uncorrupted computers can fall victim to social engineering attacks such as official-looking physical mailers with false voting codes and false voting server information.[1] Judicious selection of the distribution mechanisms for credentials and challenge mechanisms for votes may help mitigate these issues, at least to the extent that they can make it more difficult for non-malicious voters to compromise their own security.

It is very difficult—or perhaps impossible—to defend an E2E-VIV system against individual voters who are determined (or compelled) to compromise the security of their own votes; "shoulder surfing" and "rubber hose" attacks[2] remain feasible in any unsupervised voting environment, regardless of the credential distribution mechanism, voting mechanism, or other protocols implemented by the system.

Trusting the Cryptography. One area where many purportedly secure systems fall short is their cryptography; even with good intentions and sound designs, it is easy to make small but critical mistakes in cryptographic algorithm and protocol implementations that completely compromise system security.

It is clearly essential that proofs be carried out for all novel cryptographic protocols used in an E2E-VIV system. These proofs may be carried out on paper, or may be mechanized such that they are checkable by computers. Moreover, all cryptographic protocol implementations must be verified against their specifications; if the specifications are mechanized, it is easier to carry out this verification, and is even possible to *synthesize* the implementation directly from the specification in a correctness-preserving fashion.

It is also essential that only well-studied secure cryptographic algorithms and pseudorandom random number generators (PRNGs) be used in E2E-VIV systems. The need for secure cryptographic algorithms is obvious; the fact that bad sources of randomness can cause many vulnerabilities in the otherwise-sound cryptographic systems built atop them is less so. Exploits related directly to weak or absent random numbers have taken place in various real world systems, including fraudulent transactions in European EMV (Chip and PIN) payment systems, large-scale theft from Android-based Bitcoin wallets, and a complete compromise of the PlayStation 3 game console's digital signature process.

[1] Social engineering attacks have often been used in non-Internet election scenarios, primarily with the goal of suppressing votes; durng the 2014 midterm election cycle, there were numerous reports of mailings directing voters to incorrect voting places, giving incorrect election dates and absentee ballot submission deadlines, and containing incorrect information about voter registration.

[2] These are both actual terms of art in the field of computer security. A "shoulder surfing" adversary monitors all the actions of the voter throughout the voting process, while an adversary employing a "rubber hose" attack extracts information from the voter or forces the voter to take particular actions by means of threatened or actual physical harm.

E2E-VIV systems may use cryptographic algorithms and PRNGs approved by government agencies, such as those specified by NIST in the Federal Information Processing Standards (FIPS), but should be restricted to those that have also been extensively studied by non-government entities to counter the appearance that the government has a "back door" allowing it to defeat the system's cryptography.[3] Similarly, hardware entropy generators such as the RdRand functionality found in current Intel processors can be used to provide entropy in an E2E-VIV system, but must not be the sole sources of entropy; since they are opaque subsystems whose functional details are not available for inspection, they may contain hidden weaknesses or back doors. The output of such generators can be combined with other entropy as seed input to other open-source secure PRNGs, as is standard practice in most operating systems' random number generation subsystems.

Trusting the Toolchain. Ken Thompson, in his 1984 Turing Award lecture "Reflections on Trusting Trust" [190], demonstrated a fundamental problem with trust in computing systems: an attack against the toolchain (compilers, assemblers, linkers) used to build a system can silently, and effectively undetectably, insert a "back door" or other corruption into the system. If this attack is carried out successfully, inspection of the source code for the toolchain itself and the source code for the system will show nothing unusual; the corrupted toolchain binary introduces the corruption when building itself, or when building the rest of the system, and also corrupts all the tools that can be used to analyze the system (disassemblers, binary dump tools, etc.) such that the corruption remains hidden. Thompson himself successfully carried out such an attack within Bell Labs, and similar attacks have occurred "in the wild" against systems such as the Delphi development environment for Windows application; with stakes as high as controlling national election results, it is not a stretch to believe that such attacks would be attempted against E2E-VIV systems.

There are multiple ways to mitigate the possible impact of such an attack. One is to ensure that the system uses a diverse set of implementations of key components, all based on the same specification but with different source code, built with different compilers, and preferably running on different hardware and OS platforms; corruption of a single component, or even a small number of them, could then be detected by the uncorrupted components, and the effort required to corrupt the system as a whole would be much higher. Another is to counter the possibility of Thompson-style exploits by using multiple toolchains in the technique proposed by David A. Wheeler in his Ph.D. thesis, "Fully Countering Trusting Trust through Diverse Double-Compiling" [205].

6.1.6 Scalability

An E2E-VIV system, particularly at the national level, must be able to handle a wide range of demand. It is human nature that many voters will wait until the last day, or even the last hour, of a voting period to cast their votes. Moreover, it is likely that attacks against the system—and thus, system activity in general—will increase in intensity as the end of a voting period approaches. Thus, while the system may see very little sustained activity for much of an election period, it must be able to scale to extreme levels of activity at peak times. The architecture must take this into account, so that the system can be dynamically deployed on more computing and network resources as need arises. This might be done either by utilizing public cloud resources that support elastic demand or by using private resources that can be brought on- and offline as required.

6.1.7 Availability

E2E-VIV systems must exhibit high availability; Chapter 4 stated an explicit requirement for 99.9% uptime during election periods and the ability to recover from generalized (i.e., not caused by natural disaster or malicious attack on the system) failures in under 10 minutes, and higher availability—including in the face of malicious attack—would be preferable. There are several techniques for ensuring high availablity of systems, including the use of services like those provided by Cloudflare to handle traffic spikes and distributed denial of service attacks. The system architecture should be constructed in a way that does not foreclose the use of such techniques.

[3] As reported by the New York Times in late 2013, the NSA actually did insert a back door into the NIST-approved Dual_EC_DRBG PRNG (it has since been removed from the approved PRNG list) and has actively worked to insert similar back doors into other cryptographic systems.

6.1.8 Usability

Usability, including accessibility for disabled voters, is of paramount importance in an E2E-VIV system. Especially for the voter-facing parts of the system, the choice of implementation technology may have a significant effect on usability. Essentially, choices may need to be made between using Web technologies, which have significant advantages in terms of reach (cross-platform, able to be used on various sizes of device), and native applications, which tend to exhibit richer interaction design and support more accessibility features. The architecture might also allow for both types of implementation, potentially at the cost of additional architectural complexity.

6.2 Architectural Feature Model

As we have seen, there are many considerations to take into account when making architectural decisions about an E2E-VIV system. Here, we model the various architectural dimensions, and the (possibly wide) range of choices within each, to give a sense of the potential solution space for a workable E2E-VIV system architecture.

The Business Object Notation diagram in Figure 6.1 shows the architectural choices that need to be made; for each attribute of the architecture, a list of possible choices is provided. There are seven dimensions, each of which can take on a set of values. The values are chosen from 2-element sets for four of the dimensions, from a 3-element for one, and from 4-element sets for the remaining two. Since each dimension must have at least one selection (the empty set is disallowed), this yields a total of $(2^2-1)^4 \times (2^3-1) \times (2^4-1)^2 = 127{,}575$ possible architectural variants.

The architectural dimensions we have identified are the following:

- **Distribution of Authority** Authority in the system—that is, the "official" set of data stored in the system and control over access to and manipulation of that data—can be centralized or distributed. Centralized authority eliminates concerns about data consistency, as data can only be manipulated by one entity, but may cause issues related to system responsiveness, availability, and reliability. Distributed authority eliminates a single point of failure at the expense of needing to ensure data consistency and integrity. It is also possible to implement a hybrid authority model, where authority is concentrated in a small set of entities relative to the system as a whole; this is technically distributed authority because it is spread across entities, but behaves like centralized authority from the perspective of most of the entities in the system.

- **Cryptographic Protocols** The set of cryptographic algorithms and protocols used to protect voter privacy and insure ballot integrity is a critical component in any E2E-VIV system. Security characterizations of individual cryptographic algorithms (e.g., block ciphers such as AES, standard public key cryptosystems such as RSA, threshold cryptosystems such as ElGamal) can be found in the large body of cryptography literature, and the selection of individual algorithms is generally a matter of picking an appropriate algorithm and security strength for a given task. However, since novel cryptographic protocols such as those required to implement E2E-VIV systems are not widely used or studied, evidence must be provided for the security of any such protocols used in a deployed system. Such evidence can be provided in two basic ways: through "paper" proofs that are carefully checked by multiple experts, or, if the protocols are mechanized in a formal specification language, through proofs that are automatically generated by cryptography protocol verifiers such as ProVerif [28]. In general, it would be preferable for all cryptographic protocols in the system to be mechanized, as evidence could be generated repeatably and easily regenerated in the event of minor protocol changes; however, there may be cases where this is impractical. Thus, the cryptographic protocols of the system may have "paper" specifications, be mechanized in a formal system, or some combination thereof. Moreover, their implementations may be verified against the specifications in various ways, or may be directly synthesized from the specifications in a way that guarantees correctness.

- **Evidence of Correctness** The development of a high assurance system such as an E2E-VIV system proceeds from a specification, at some level of formality, to an implementation that is intended to fulfill the specification. Assurance that the implementation actually does fulfill the specification can generally be obtained in two ways. First, the implementation can be developed in a way such that it is mechanically tied to the specification; for example, code generation techniques and refinement techniques can be used to mechanically

```
static_diagram E2EVIV_Architecture_Dimensions
  -- This diagram shows the various dimensions of an E2EVIV architecture
component
  class E2EVIV_ARCHITECTURE
  feature
    authority_distribution: SET[VALUE]
      ensure 0 < Result.count;
             for_all v: VALUE such_that v member_of Result
               it_holds v member_of { Centralized, Distributed };
      end
    crypto_protocols: SET[VALUE]
      ensure 0 < Result.count;
             for_all v: VALUE such_that v member_of Result
               it_holds v member_of { On_Paper, Mechanized,
                                      Verified, Generated };
      end
    correctness_evidence: SET[VALUE]
      ensure 0 < Result.count;
             for_all v: VALUE such_that v member_of Result
               it_holds v member_of { Process_Based, Assertions };
      end
    implementation_type: SET[VALUE]
      ensure 0 < Result.count;
             for_all v: VALUE such_that v member_of Result
               it_holds v member_of { Golden_Implementation,
                                      Open_Protocols_and_Specs };
      end
    key_distribution_method: SET[VALUE]
      ensure 0 < Result.count;
             for_all v: VALUE such_that v member_of Result
               it_holds v member_of { Public_Ceremony, Threshold_Cryptography,
                                      PKI, Web_of_Trust };
      end
    deployment_style: SET[VALUE]
      ensure 0 < Result.count;
             for_all v: VALUE such_that v member_of Result
               it_holds v member_of { Trusted_Servers, Public_Cloud, Peer_to_Peer };
      end
    client_technology: SET[VALUE]
      ensure 0 < Result.count;
             for_all v: VALUE such_that v member_of Result
               it_holds v member_of { Custom_App, Web_Based };
      end
  end
end
```

Figure 6.1: A specification of the possible variants for an E2E-VIV system.

generate correct implementations from specifications. Second, the implementation can be developed "by hand" with a set of included assertions that are meant to establish that the specification is being faithfully implemented; these assertions may then be checked by code analysis tools when building the system, or may be automatically compiled into testing code that validates the assertions when running the system. In practice, while it is clearly desirable for as much of the implementation as possible to be mechanically generated from the specification, most high assurance systems use some combination of these two techniques.

- **Implementation Type** Regardless of the choices made along any of the other dimensions, a reference for what constitutes a "correct" implementation must be provided. Either a "golden implementation" with strong correctness guarantees or a complete set of open protocols and specifications may serve as such a reference.

  A golden implementation G may be synthesized directly from a formal specification using semantics-preserving transformations, may be implemented in one of several languages using a "correct by construction" approach, or may be painstakingly verified against a formal specification either by hand or in an automated fashion. In any case, such a G defines the correct behavior of the system but may—because of its implementation technology or other factors—not satisfy real-world performance requirements, memory usage/storage requirements, etc. However, G can be used as a reference point for other implementations; if the behavior of an implementation X, which does satisfy deployment requirements but for which we have no strong correctness guarantees, conforms to that of G, then X is also a correct implementation (and is likely better suited for deployment).

  In the absence of a golden implementation, evidence for implementation correctness can come in the form of conformance testing against open protocols and specifications. This requires that such protocols and specifications are provided for the entire system in formats such that conformance checking is feasible.

- **Key Distribution Method** In all the possible implementations of an E2E-VIV system, encryption keys will be required to fulfill security, privacy, and integrity requirements. The ways in which these keys will be generated is determined by the cryptographic algorithms chosen for use in the system, but the ways in which they are distributed depend on architectural choices. For example, public "key generation and distribution" ceremonies and the use of threshold cryptography for the keys that enable generation of the final tally, described in Section 6.1.5, may be used to improve trust in the system and making it more resilient in the face of attempts by the electoral authority to determine votes or prevent the completion of the election tally or its audit. However, the generation and use of encryption keys in the system is not limited to the election authority; as a result of the pervasive use of TLS encryption for communication, keys are continually generated and exchanged throughout the system.

  The most widespread method of key distribution today is the public key infrastructure (PKI), which manages digital certificates and related artifacts. Trusted certificate authorities issue certificates that bind public keys with user or organizational identities; this is what allows a user to connect to her bank online, examine the certificate presented during the connection, and see that the certificate was in fact issued to her bank rather than a man-in-the-middle attacker. As described in Section 6.1.5, PKI can be used to guarantee to voters (in the event that they check) that their connections to an E2E-VIV system are secured using a certificate issued to and signed by the electoral authority. PKI is effectively mandatory for secure Internet connections using the TLS protocol, but may also be used to manage other keys in the system; however, this requires a trusted certificate authority for such keys.

  Another method of key distribution is the "web of trust", first described by the creator of Pretty Good Privacy (PGP) [212] in 1992. Unlike a typical PKI system, a web of trust system does not have the concept of trusted certificate authorities; instead, it is essentially a large collection of public keys that can be digitally signed as a signal of trust. Each public key in the repository identifies an entity (individual or organization) that possesses the corresponding private key. Each public key can be digitally signed by the owners of other keys. For example, assume Alice and Bob have both published their public keys, which are A and B respectively. If Alice knows Bob personally, and Bob can demonstrate to Alice that his public key is actually B, then Alice can sign key B (with her private key) to indicate her trust that key B belongs to Bob. Such signatures build a distributed set of trust relationships: if Alice trusts Bob, and Bob trusts Charlie, then it is presumed that Alice can trust Charlie. This is similar to the way certificate authorities work in PKI systems; in effect, Bob acts as a

certificate authority for Alice because of the trust relationship they have established, and Alice can trust any key that he has signed (or that has been signed by a key he has signed, etc.). Key distribution via a web of trust could be useful in various parts of E2E-VIV systems, as a means of collectively building trust relationships within the system rather than relying on the single points of failure inherent in PKI infrastructures.

- **Deployment Style** The E2E-VIV software can be deployed in one of three ways: (1) servers run entirely on trusted servers managed by the electoral authority or its designated representatives, and client applications access the servers through well-defined, well-controlled interfaces; (2) servers run, in whole or in part, on public cloud infrastructure, while client applications still access them through well-defined interfaces; (3) the system is structured in a peer-to-peer fashion, where "server" functionality is distributed across all entities in the system and at least some of them run in an uncontrolled environment (i.e., voters' computers).
- **Client Technology** Implementation of the client software used by voters, as well as the administration software used by the electoral authority, can be done in two basic ways: (1) develop custom applications for the various hardware/OS platforms that will be used by voters and the electoral authority; or (2) use Web application technologies to develop a single Web-based application that will be accessible from all (reasonable) platforms. It is also possible to choose both implementation strategies for all applications (i.e., both voters and the electoral authority can access the system through either native applications or a Web application, as they choose) or make different choices for different applications (e.g., the voter application is implemented as a Web application while the electoral authority's administration application is implemented as a native application).

6.3 Primary Architectural Variants

Given the many dimensions of the architectural feature model, and the number of choices in each, the system has many possible architectural variants. Here, we briefly discuss a few of the primary system variants that can be described by the feature model. Since we are only describing them at a high level, some of the variants correspond to many possible feature selections in the feature model (for example, they might have any of the different types of correctness evidence or any of the different key distribution methods).

6.3.1 Mirrored Servers

One possible architecture, which features centralized authority as its primary defining characteristic, is the "mirrored servers" architecture depicted in Figure 6.2. The double arrows in this diagram (and later diagrams in this chapter) denote *client* relationships (one entity making use of another's services), while the thick double-ended arrows denote *mirroring* relationships (entities, or groups of entities, ensuring that their states accurately reflect each other for redundancy or availability).

In this architecture the Web/App Server (to which voters, using either a web-based interface or custom applications, connect to cast their ballots) is a client of the Database, which stores all information relevant to the operation of the E2E-VIV system (ballot styles, cast and spoiled ballots, etc.). In an actual implementation, the monolithic Database would likely be split into multiple databases since the access patterns and performance needs for data such as ballot styles, cast and spoiled ballots, voter lists, etc., are likely to be quite different. There might also be more components within each mirror (for example, separate servers for dealing with native applications vs. web access in a system that supports both).

Regardless of the number of servers within each mirror, the mirroring in this architecture is done primarily for availability and reliability; it ensures that, as long as at least one set of mirrored servers is running, the system can remain operational (albeit perhaps at a degraded level of responsiveness). Authority is centralized in the sense that each mirror has a complete set of data for the system and behaves accordingly; one mirror is designated as the *primary* mirror and is considered the authoritative source of information in the event of inconsistency. Voters and the electoral authority access the system by interacting with an individual (typically, the primary) mirror, and the entire set of mirrors appears logically as a single server-side system.

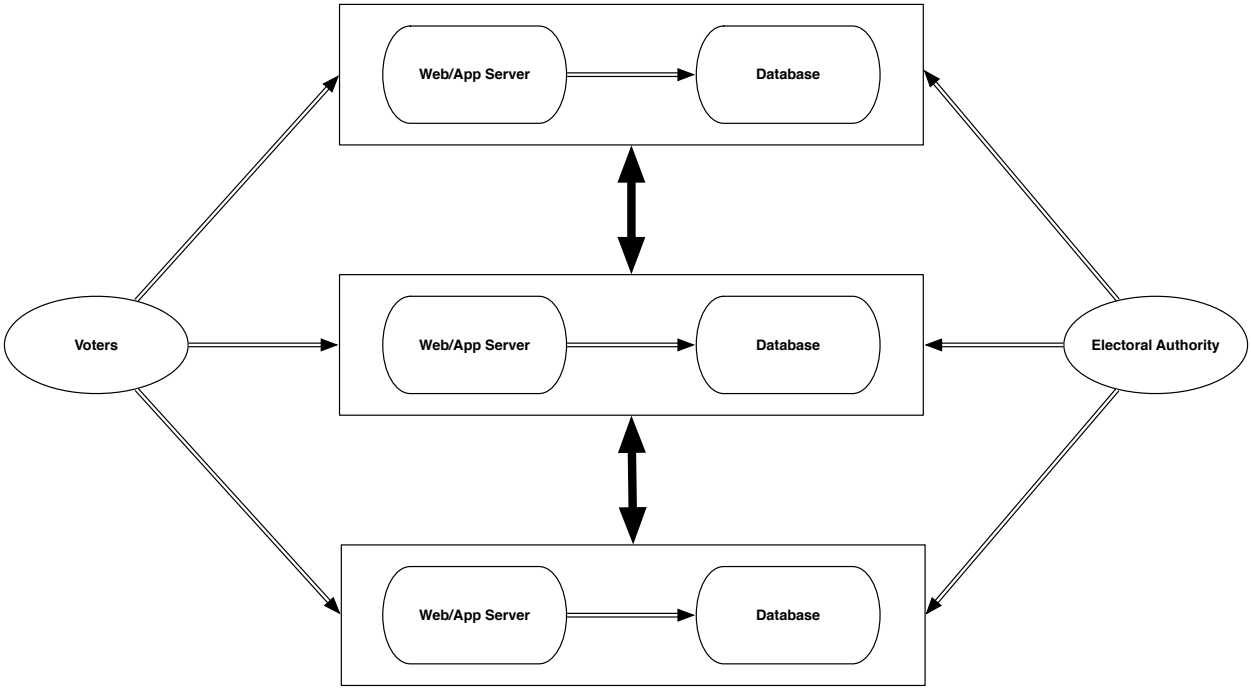

Figure 6.2: An architecture with centralized authority and mirrored servers.

6.3.2 Large Fixed Set of Servers

Another possible architecture, which introduces the potential for distributed authority but still has the logical presentation of a single server-side system, is a large fixed set of servers. This architecture, an example of which is depicted in Figure 6.3, still features mirroring for redundancy and availability; however, it allows for flexible allocation of resources. For example, there might be twice as many Web/App Server instances as there are Database instances, or there might be more Database instances dealing with dynamic cast and spoiled ballot data than dealing with fixed election definition data such as ballot styles.

The servers within this architecture could, amongst themselves, behave as a peer-to-peer system, a set of client-server systems, or a set of mirrors of various sizes for the purposes of providing high availability and redundant storage and ensuring data consistency. A key aspect of this architecture is that the number of servers, while large, is fixed; this allows the topology of the servers and the communications amongst them to be known at all times, making it straightforward to monitor the system's health and performance and to quickly detect any issues that arise.

As an example, Figure 6.3—only one of many possible server topologies in such an architecture—has two separate mirrored Databases (one with two mirrors, and one with three) being accessed by three separate Web/App Servers. If it is determined that the Databases are underloaded and the Web/App Servers are overloaded, one of the servers running Database B could easily be repurposed to run an additional Web/App Server (Figure 6.4) without changing the actual set of servers in the architecture and without compromising the redundancy of data storage in the system.

While one possible deployment of this architecture would see every server containing the full authoritative data set, it is far more likely that each would contain only part of it and that the authority in the system would, therefore, follow either a hybrid or a distributed model.

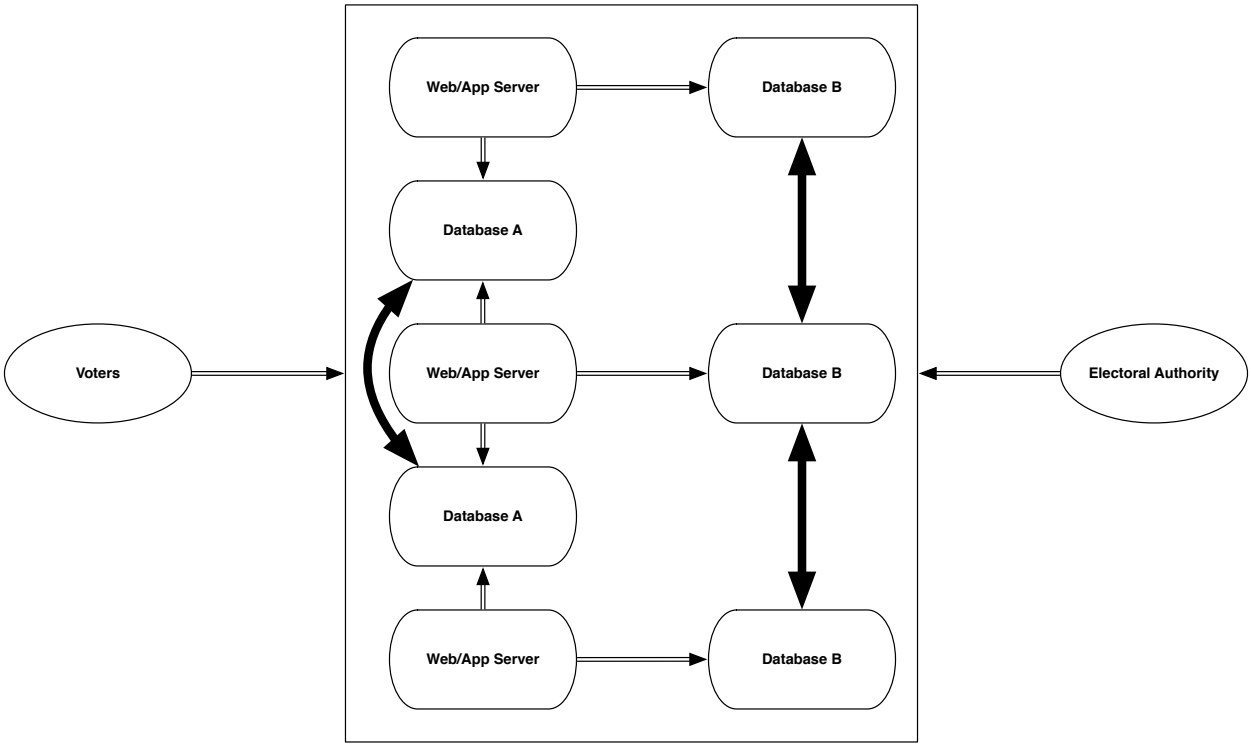

Figure 6.3: An architecture with a large fixed set of servers.

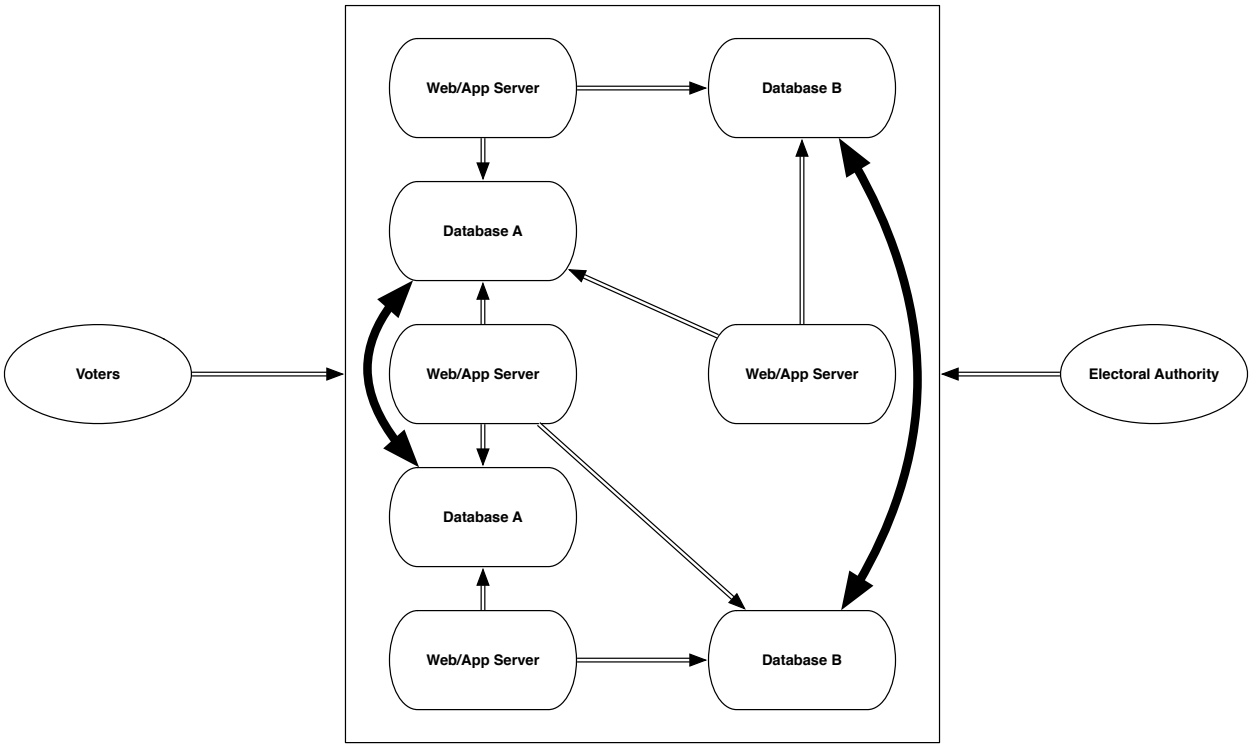

Figure 6.4: The same fixed set of servers as in Figure 6.3 performing a different allocation of tasks.

6.3.3 Dynamic Cloud

The two previous architectural variants involved the deployment of a fixed set of servers, either as a collection of mirrors or in other topologies. The next variant departs from these by deploying services not across a fixed set of servers, but instead within a dynamic cloud infrastructure, while still presenting itself as a single server-side system for external interactions. Such an infrastructure allows for the addition and removal of computing resources as necessary during the operation of the system, using various distributed communication and consistency protocols to deal with resource changes in a way that is effectively invisible to the system's users while maintaining data integrity and service availability. Figures 6.5 and 6.6 show snapshots of a dynamic cloud deployment at times when it has five and eleven running servers, respectively. Note, in particular, that the client relationships among the servers in the cloud may evolve over time as well; for example, in Figure 6.5, the server at the "top" of the cloud could establish direct communication with the server at the "bottom left" of the cloud if necessary.

Effectively, a dynamic cloud deployment behaves similarly to a deployment with a large fixed number of servers; the main difference is that the number of servers is variable. This allows for the system to initially consume minimal resources, expanding or contracting as necessary (within the bounds of the dynamic cloud) to maintain acceptable response time and availability in the face of elastic demand.

Despite the use of the word "cloud", a dynamic cloud architecture need not actually be deployed on a public cloud infrastructure; private cloud infrastructures consisting of only trusted servers may be built as necessary to support the system.[4] Regardless of whether the system is deployed on a trusted or public infrastructure, authority in a dynamic cloud architecture follows either a fully distributed or a hybrid model; some servers in the cloud may have authority over others, or they may interact using consensus protocols or similar mechanisms.

6.3.4 Peer-to-Peer

In all the architectural variants described so far, the system presents itself as a single "server" regardless of its "internal" network topology. In a *peer-to-peer* implementation, the computational work of the system is distributed across all the participants and there is no clearly defined distinction between "client" and "server". For example, Figure 6.7 depicts a peer-to-peer system with some peers belonging to individual voters, some belonging to political parties (A, B and C), and some belonging to the electoral authority. The double-headed arrows in the figure represent communication links among the peers; for example, if the upper-left peer belonging to Party A needs to communicate with the lower-right peer belonging to the electoral authority, it must send a message that travels across at least 7 communication links. The communication links in a peer-to-peer network typically change over time, based on each peer's knowledge about its network environment and the locations of other peers.

Authority in a peer-to-peer architecture is fully distributed. In the case of an E2E-VIV system, the electoral authority would set up and maintain some trusted peers as a way to "bootstrap" the peer-to-peer network, and political organizations (parties, lobbying groups, etc.) might also choose to maintain peers, perhaps with their own implementations of the election software in a system designed with open protocols and specifications, as a way of participating in the electoral process and strengthening trust in the results. Individual voters running the software on their own machines would also be peers for the duration of their voting sessions (or longer, if they chose to contribute to the management of the election by leaving the software running); effectively, a peer-to-peer architecture is a way of "crowdsourcing" the resources required to run election system.

[4] In the E2E-VIV context, however, public cloud infrastructures are likely preferable for economic reasons; it is virtually inconceivable that an electoral authority or its suppliers could build a cloud infrastructure with scale and reliability comparable to existing public cloud infrastructures at reasonable cost.

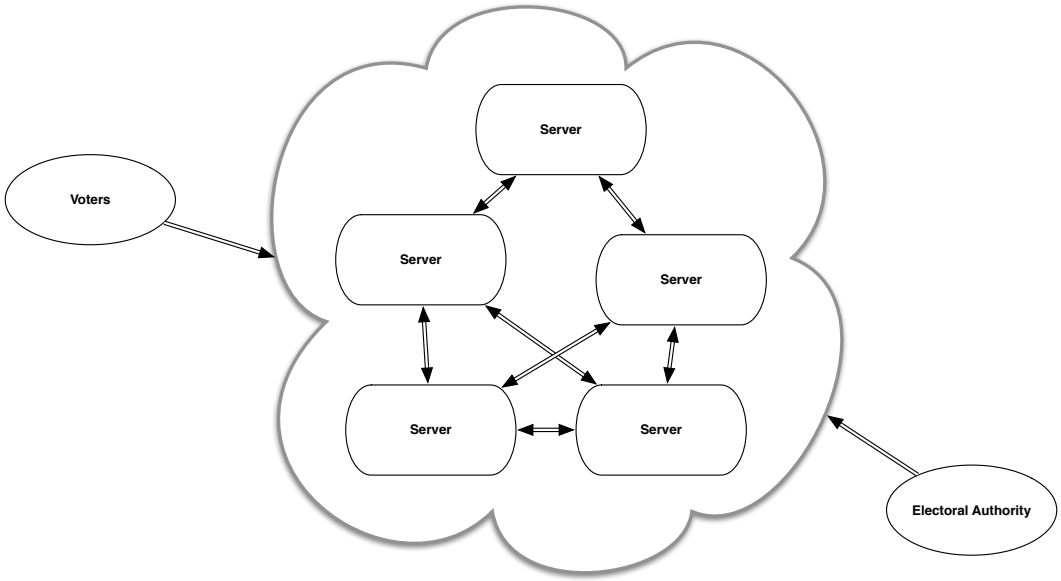

Figure 6.5: A dynamic cloud architecture with a small number of servers.

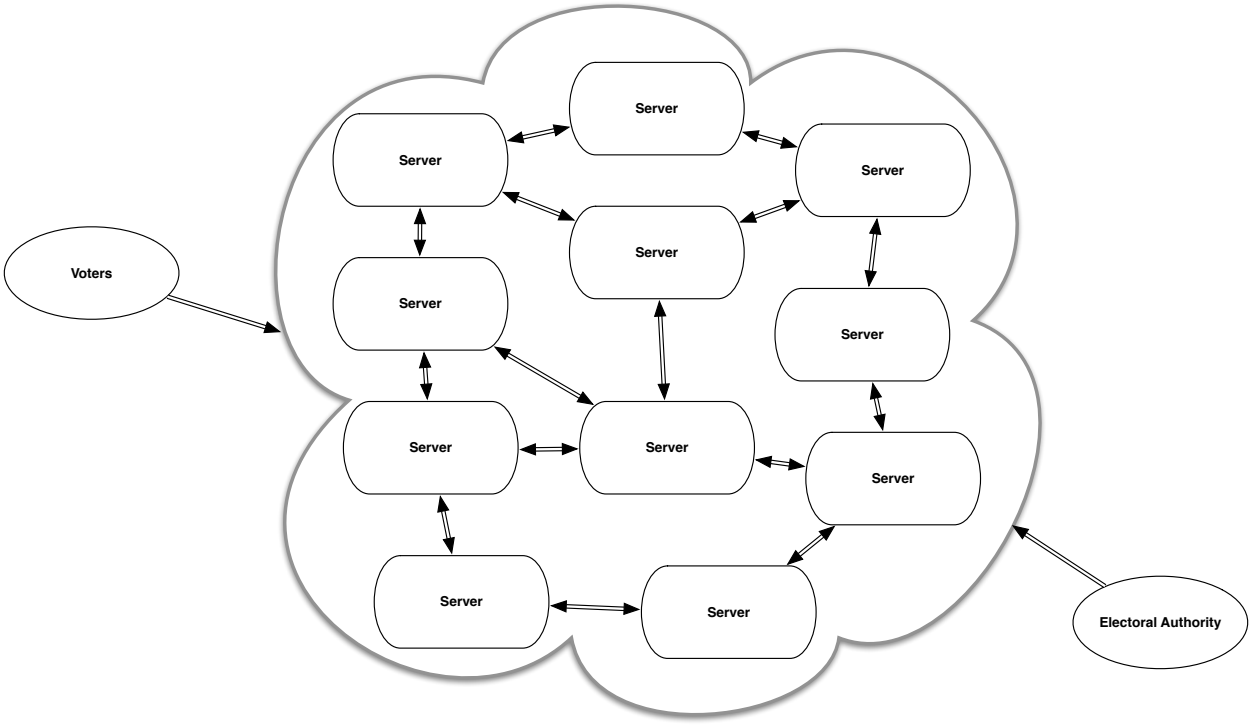

Figure 6.6: A dynamic cloud architecture with a larger number of servers.

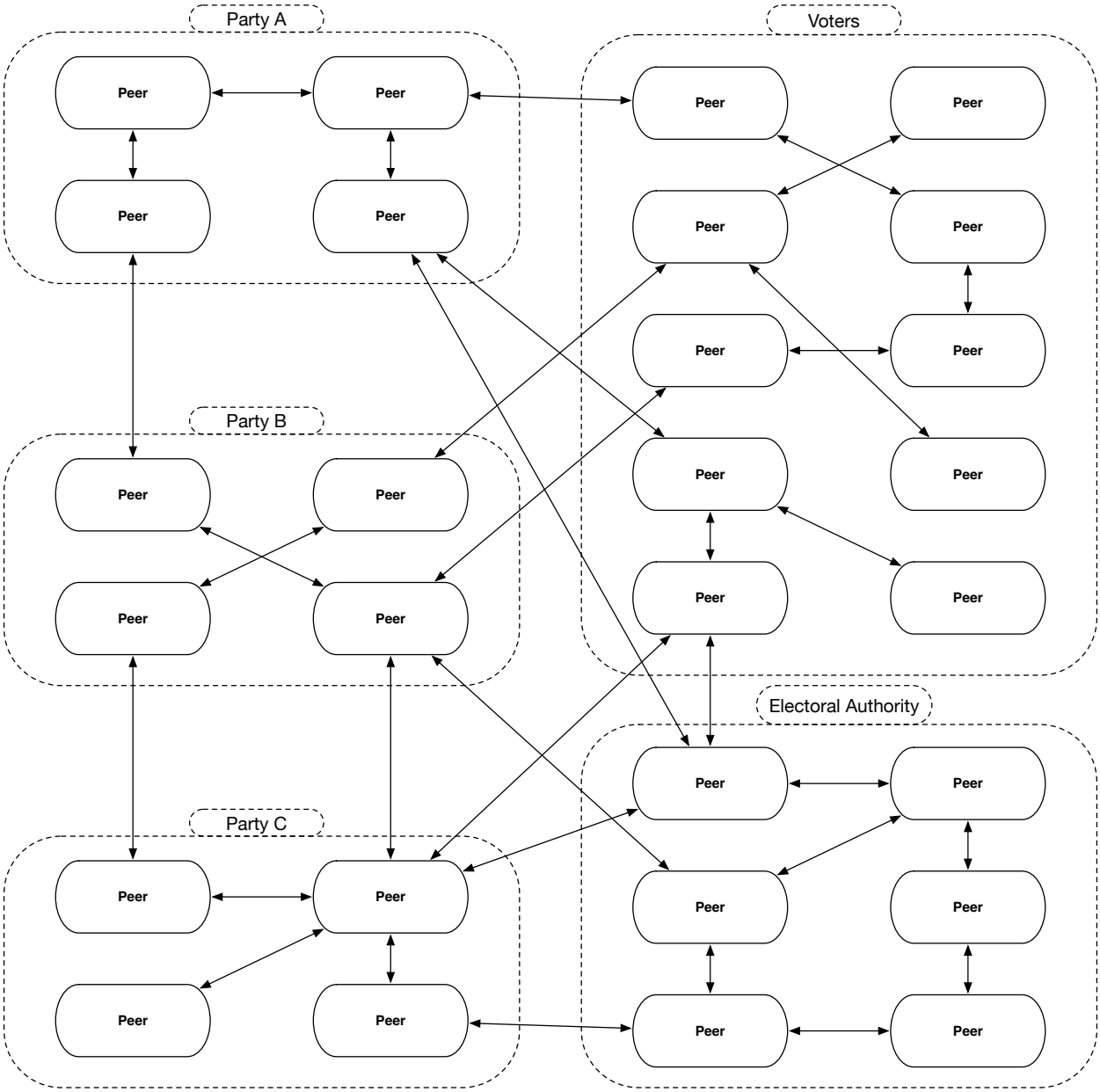

Figure 6.7: A peer-to-peer architecture.

A peer-to-peer architecture raises significant security concerns that differ from those of the other architectures we have described. While some of the computer systems controlled by the electoral authority might be trusted, the vast majority of systems belonging to individual voters or political organzations will certainly not be; it is impossible to run a peer-to-peer E2E-VIV system using only trusted computing resources.[5] It is therefore important to ensure that no corrupt peer, or set of corrupt peers, can undetectably compromise election results, violate voter privacy, or otherwise violate the E2E-VIV system requirements.

[5]This is true of "pure" peer-to-peer architectures of the type we are discussing in this section; an architecture where only the *servers* interact in a peer-to-peer fashion while presenting a single interface or set of interfaces to clients can, as previously noted, be implemented on a fixed set of servers or in a dynamic cloud.

One way to address this problem is to employ a *blockchain*, like that used in Bitcoin and other cryptocurrencies, to log critical election information (cast and spoiled ballots, the fact that a given voter has voted in the election, etc.). A blockchain is a public write-only ledger, collectively maintained by the peers in the system, that records a sequence of events. The mechanism by which this recording is done ensures that the peers reach a consensus about the events that have occurred and their ordering, and that once an event (such as the casting of an encrypted ballot) has been placed in the ledger it can be neither modified nor reordered with respect to other events. As long as more than half of the peer computing power in the network is "honest" and follows the correct protocol, the integrity of the ledger is guaranteed. At any given time, it is likely that the computing power contributed by the electoral authority and high-profile political organizations—which can be hosted on trusted, closely-monitored computing systems—will vastly outweigh the computing power contributed by individual voters during their ballot casting sessions; moreover, the situation where more than half the peer computing power is dishonest can be detected (by the honest part of the network, or by external observers) and dealt with in various ways. Thus, maintaining the integrity of a blockchain should be reasonably straightforward in an E2E-VIV system. However, other aspects of implementing a peer-to-peer architecture—such as distribution of the computing client to voters and organizations, achieving sufficient ease of use and performance, etc.—may prove more difficult.

6.4 Summary

As can be seen from the many architectural dimensions we have described and the primary architectural variants we have briefly discussed, there are many different ways in which an E2E-VIV system might be designed and implemented. We don't yet know which, if any, of these approaches could be used as part of a system that could satisfy the broad range of requirements for an responsible Internet voting system. Significant additional research and experimentation would be necessary to determine if a suitable path forward can be found for an E2E-VIV implementation and deployment.

Chapter 7

Rigorous Software Engineering

Sound engineering practices are the foundation for building reliable and secure software systems. This is particularly true for critical systems that must be trusted to perform important tasks correctly, and where the consequences for failure threaten life, property, or political system integrity.

The engineering of any software system can be roughly divided into four stages: specification, implementation, verification/validation, and maintenance. These stages are not mutually exclusive; for example, a specification can be refined during implementation and verification/validation can occur continuously during implementation and maintenance.

Specification is the most important stage, particularly when building critical systems: it informs the entire development process by determining what must be built and what it must, and must not, do. Even if a system works perfectly, it is impossible to provide convincing evidence of its correctness without using and consistently maintaining a high-quality, accurate specification throughout the development process. Such evidence is essential for critical systems to which we entrust our lives, money, or votes.

Implementation is the transformation, either automatic or manual (typically some of each), of a system's specification into executable code. The quality of an implementation is heavily influenced by choices of programming languages, coding conventions, and supporting tools, some of which are driven by choices made during specification.

Verification and validation are closely related, but independent, procedures. Verification is an evaluation of whether the implementation is a correct realization of its specification, while validation is an evaluation of whether the implementation is satisfactory to external stakeholders. Verification and validation are typically carried out continuously and automatically during the development process, using a variety of tools and techniques; the results of verification and validation are the primary sources of evidence for the implementation's correctness.

Maintenance includes tracking and fixing defects discovered in the implementation, modifying the specification and implementation as required by technical issues or feedback from external stakeholders, and adapting the implementation to new hardware and software platforms and new deployment environments. Maintenance is closely linked with verification and validation, since new evidence of the system's correctness must be provided when changes are made to either its specification or its implementation.

This chapter introduces rigorous software engineering techniques for these stages of the software development processes. We particularly focus on specification, verification, and validation, since implementation and maintenance are widely discussed throughout the software engineering literature; however, we do describe specific implementation and maintenance techniques that are particularly important when developing critical systems. Finally, we recommend specific technologies to use when constructing critical systems and briefly discuss the current state of elections technology in light of our recommendations.

7.1 Informal Specifications

An *informal system specification* is a human-readable description of the system's purpose, functionality, and high-level design. Informal specifications should be understandable not only to the system's developers, but also to other stakeholders: clients for whom the system is being developed, users of the system, administrative staff who maintain the system, and auditors who evaluate the system.

A complete informal system specification consists of a domain model, a set of requirements and scenarios, and a set of concept specifications.

7.1.1 Domain Models

A *domain model* of a system identifies the various concepts (also called *classes* or *classifiers* in some domain modeling techniques) in the system, their attributes and roles, and the client and inheritance relationships among them. Domain models can be expressed in various ways, both textual and graphical. Regardless of their form, they are essentially lists of concepts with associated brief human-readable descriptions of themselves and their relationships to other concepts. A good domain model provides a shared vocabulary and gives a big picture view of the concepts involved in the system upon which all the stakeholders can agree.

For example, the domain model for E2E-VIV systems [125] includes, among many others, the concepts "election", "contest", "ballot question", and "choice". It describes an election as "a formal process of selecting choices in one or more contests", and says that a ballot question is a specific type of contest, namely "a decision among two or more courses of action". The relationship between a ballot question and a contest is an inheritance relationship: a ballot question *is a* specific type of contest, with a certain set of characteristics. One relationship among elections, choices, and contests is that an election *has*, or *is a client of*, at least one choice and at least one contest, because the description of an election explicitly refers to choices and contests.

Importantly, while a good domain model comprehensively identifies and defines the concepts and relationships in a system, it does not otherwise constrain the realization of those concepts and relationships or the actual behavior of the system. The description of "ballot question" does not specify *how* a decision among two or more courses of action is made, and the description of "election" does not specify *how* choices in contests are selected. Such constraints on the realization and behavior of the system are described in general terms in requirements, scenarios, and informal concept specifications, and are precisely expressed in formal specifications.

7.1.2 Requirements and Scenarios

A system's *requirements* are, essentially, the statements that must be true of the system's implementation when it is complete. Requirements are typically phrased as simple natural-language sentences, each of which expresses a single testable property of the system. "The e-voting system shall maintain the privacy of individuals." and "All voter authentication secrets must be changed at least once in every election cycle." are examples of E2E-VIV requirements; the full set of requirements is described in Chapter 4, and the informal requirements themselves are available in a separate document [126].

There are several different kinds of requirement, but they can be broadly divided into two basic types: *functional* and *non-functional*. Functional requirements are those that specify how the system must behave and what outputs it must generate given certain sets of inputs under certain conditions. For example, one functional requirement of E2E-VIV systems is that only eligible voters may cast ballots. Non-functional requirements are those that specify overall characteristics of the system, such as a lower bound on its availability or an upper bound on its cost. For example, one non-functional requirement of E2E-VIV systems is that they must be Open Source.

Scenarios are descriptions of interactions among the entities in the system, or between the system and its environment. A scenario is typically expressed as a short paragraph describing the actions of the entities involved, though numbered lists (to indicate a sequence of operations) or communication diagrams (to indicate the flow of data during a scenario) are also reasonable representations. A scenario may also be expressed as a collection of requirements that, together, describe the behavior of a part of the system under certain circumstances.

It is important that requirements and scenarios describe not only the *ideal* behavior of the system, but also how the system deals with situations that are less than ideal. System failures, communication disruptions, data corruption, and malicious attacks of various kinds should all be addressed in a system's requirements and scenarios. It is not necessary to specify these at a level of detail that provides explicit sequences of steps to recover from any given failure or attack; rather, they can be addressed in general terms. For example, one requirement of E2E-VIV systems that addresses system failures is "If service goes down for any reason other than regional natural disaster or malicious attack, service must be restored in no more than 10 minutes".

As the system is built, *traceability* of the requirements and scenarios is critical. It must be possible to identify, for every requirement and scenario, the related parts of the formal specification and implementation, and the links among informal specification, formal specification, and implementation need to be kept current when any of them change. Moreover, it must be possible to identify one or more system tests demonstrating that the system implementation adequately addresses each requirement or scenario. This should preferably be done automatically as part of testing and continuous integration processes, in a way that makes it clear to developers which, if any, requirements and scenarios remain to be addressed.

7.1.3 Concept Specifications

A *concept specification* is an informal behavioral description of a concept in the system's domain model. Concept specifications help to clarify what the roles and responsibilities of the concepts in the model are with respect to the overall functionality of the system. A typical concept specification is a set of English sentences called *queries*, *commands* and *constraints*, describing respectively what information the concept possesses, what the concept can do, and what limitations exist on its behavior.

A query is phrased as a simple question, such as "How many votes have been cast?" or "Is this voter eligible to vote in this contest?", that the concept must be able to answer. Queries provide an informal description of the data encapsulated in the concept, because the concept must contain all the data necessary to answer any of the queries that may be posed to it.

A command is phrased as an exclamation, such as "Cast this vote!" or "Log this user out!", describing an action that the concept must take. This provides an informal description of the behavior encapsulated in the concept, because the concept must be able to take all (and only) the actions described by its commands.

Finally, a constraint is phrased as a declaration, such as "A voter must be eligible for this election to cast a vote." or "Only users that are logged in may be logged out.", describing the circumstances under which queries and commands may be issued. This provides an informal description of the conditions under which queries and commands may be issued to a concept.

An informal concept specification has two main purposes: first, it clarifies the roles, responsibilities and relationships of a concept in the system. The process of creating informal concept specifications may lead to the realization that some necessary concepts are missing from the domain analysis, that some single concepts are actually multiple related concepts that need to be described separately, or that multiple concepts that were thought to be distinct are actually aspects of the same concept. Second, it serves as a "template" for developers implementing the concept, and—with appropriate tool support for the formal specification and implementation languages—can be used to automatically generate initial formal specifications and implementation frameworks.

7.2 Formal Specifications

A *formal system specification* is a machine-readable, mathematical description of the system's functionality and design. It is a refinement of an informal system specification, and there must be a demonstrable and traceable correspondence between the informal and formal system specifications. Formal specifications are meant to be used both by the system's developers and by the various tools employed during the development and testing process to validate the system against its specification. They may also be used to automatically generate partial or full implementations of system components, create and run test suites, and generate part of the system documentation.

A complete formal system specification consists of architecture, concept, source code, and protocol specifications; some of these, such as the architecture specification, may be minimal for simple systems. As we describe these types of specification, we also introduce the verification techniques that are applicable to each.

7.2.1 Architecture Specifications

An *architecture specification* is a precise description of the system's architecture, formalizing both the relationships among the concepts in the system and the relationships between these concepts and the physical implementation of the system.

Architecture specifications describe both the software architecture and the hardware/network architecture on which the software runs. A software architecture specification describes how many instances of each concept exist within the system, the communication patterns among the concepts, and any containment relationships among the concepts. It also describes, at least at an abstract level, parts of the overall software architecture that are external to the system under development. For example, it may describe the relationships between the concepts in the system, the services provided by the operating system on which the software will run, and any external services (such as databases or web servers) on which the software depends.

A hardware/network architecture specification describes the relationships among the various hardware components in the system. This includes the services provided by each component and the network connectivity and communication patterns among them, as discussed in Section 6.3.

Some research has been done with respect to verifying software implementations or hardware simulations against architectural specifications [1, 134] and some tool support exists for such verification; however, it is currently infeasible to verify a distributed hardware and software implementation against an architectural specification.

7.2.2 Concept Specifications

A *formal concept specification* is a formal behavioral description of a concept in the system's domain model. Formal concept specifications are refinements of informal concept specifications, as described in the previous section. It is possible for multiple informal concept specifications to refine to the same formal concept specification; for example, two informal concept specifications "list" and "queue" might both refine to the formal concept specification "linear data structure".

Like informal concept specifications, formal concept specifications have queries and commands; in formal concept specifications, these are expressed as typed interfaces to the concept called *features*. For example, a formal specification of the "election" concept might refine the informal query "How many votes have been cast?" with a feature of type `integer` called `number_of_votes_cast`, and the informal command "Cast this vote!" with a feature of type `vote -> void` (that is, taking a vote as a parameter and returning no result) called `cast_vote`. It is possible for a single query or command to refine to multiple features of different types; for example, a "Log in!" command might be implemented as two features, one taking username and password parameters and the other taking an authentication token parameter.

Features may be restricted, accessible only to a given concept or group of concepts, or unrestricted, accessible to the entire system. It is possible for a concept to have restricted features that are not direct refinements of its informal queries and commands. For example, the "election" concept might have restricted features representing the entire database of candidates for office and ways to update that database, but only allow unrestricted access to the database via a query (refined from the informal specification) requesting the candidates for a particular race.

Formal concept specifications also have constraints, expressed as *preconditions* and *postconditions* on features or as *invariants*, as appropriate. Preconditions, postconditions and invariants are part of the Design by Contract development technique [140]: a precondition is a predicate that must be true in order to invoke a feature; a postcondition is a predicate that must be true after a feature has been invoked; and an invariant is a predicate that must be true at all (observable) times. For example, an informal constraint that the number of votes cast can never be negative could be expressed as a postcondition `Result > 0` on the `number_of_votes_cast` feature (guaranteeing that the result of invoking the feature is always non-negative), or could be expressed as an invariant `number_of_votes_cast > 0` on the "election" concept as a whole.

In addition to refinements of queries, commands and constraints, each formal concept specification also contains refinements of inheritance and client relationships described in the domain model. Inheritance relationships are directly stated in formal concept specifications, and client relationships are implicitly stated through the types assigned to the concept's features.

A formal concept specification, like an informal one, has two main purposes. First, it expresses some of the requirements of the system in precise mathematical terms that can be checked for various desirable properties, such as logical consistency, before implementation begins; this allows for early detection of a class of possible problems with the requirements or their realizations. Second, it can be used with automated code generation tools to create a significant amount of the implementation and source code specification automatically, in a provably-correct and traceable fashion.

7.2.3 Source Code Specifications

A *source code specification* is an annotation integrated into, or otherwise associated with, a piece of source code that makes formal statements about the code's behavior. Source code specifications are typically direct refinements of the preconditions, postconditions and invariants from formal concept specifications, and often are the same mathematical statements rendered in a different syntax. However, source code specifications may also formalize aspects of the system that are not dealt with in the formal concept specifications, such as memory and CPU usage limits, algorithmic efficiency, fine-grain concurrency properties, and cross-concept properties that cannot be expressed (or are difficult to express) in individual concept specifications, but are otherwise expressed in, e.g., non-functional requirements.

Source code specifications are generally written in a language that is closely related to the implementation language, and in some cases are written in the implementation language itself. Thus, the choice of implementation language has a significant effect on what source code specifications can be written and how much effort it takes to write them. With appropriate tool support, which is available for several implementation languages of various styles, many source code specifications can be generated automatically from formal concept specifications. Additionally, some can be inferred by analyzing parts of the implementation.

Source code specifications enable extended static checking (ESC) [67] tools to verify, using automated theorem proving techniques, that implementations satisfy their formal specifications without actually executing the code. This verification is *modular*, meaning that individual components of the implementation are verified in isolation. For example, when verifying the postcondition of a feature X the verifier makes three assumptions: first, that X's precondition is satisfied when X is invoked; second, that all invariants applicable to X are satisfied when X is invoked; and third, that all features invoked by X behave correctly with respect to their specifications. This modularity allows static verification to be performed continuously during system development in an efficient, incremental fashion, providing assurance that the completed parts of the implementation are correct and highlighting the parts of the implementation that do not yet satisfy their specifications.

Source code specifications can provide significant benefits at runtime as well. Runtime assertion checking compilers can use the specifications to generate executable code that continuously checks for runtime violations of the specification and flags errors when such violations occur. Automated test generators can use the specifications to generate high-coverage unit test suites to exercise the implementation. Runtime assertion checking and automated test generation can significantly increase the reliability of the finished system, and in most cases require minimal developer effort after the specifications are written.

7.2.4 Protocol Specifications

A *protocol specification* is a formal description of an information exchange among multiple parts of a system. Each distinct type of information exchange in the system—such as registering a candidate or casting a vote—has an associated protocol that must be followed. We consider only *application-level protocols* that specify what type of information is exchanged, how it is encoded, how (and whether) it is encrypted or digitally signed, and the sequences of interactions the involved parties perform. We assume the existence of well-specified lower-level protocols that enable information transmission, such as the transport protocols used to encode information into Internet Protocol packets and the physical protocols used to convert those packets into electrical or optical signals and send them to remote destinations.

Application-level protocols are described, typically in terms of communicating finite-state machines, using one of several languages specifically devised for protocol specification. Automated tools process these descriptions to verify that the protocols have, or demonstrate that they do not have, various properties. These include both security properties, such as whether an adversary can gain access to data that is supposed to be secret or modify data without being detected, and non-security properties, such as whether the protocol always terminates in an acceptable state. Protocol specification languages and tools are typically designed to specify and verify cryptographic protocols, but can also be used to specify and verify insecure communication protocols by simply ignoring security properties; in the case of an E2E-VIV system, the vast majority of the application-level protocols are cryptographic protocols and require security property verification.

The choice of protocol specification language is almost always independent from any other specification language choices made when engineering a system, because of its specialized nature: specifying and verifying the interactions of a multiparty protocol is significantly different from specifying and verifying the behavior of individual features or software modules. However, once a protocol is verified, parts of its formal description can be embedded in the source code specifications of appropriate system modules. For example, the module that implements the receiving side of a vote casting protocol can (and should) contain an associated formal model of the appropriate protocol state machine and its state transformation rules; this model can then be verified and validated in a modular fashion alongside the rest of the source code specifications to provide evidence that the system actually implements the verified protocol.

7.3 Implementation Methodology

Once initial informal and formal specifications are developed, the remaining stages of the software development process can begin. In this section we describe some best practices for software implementation, validation and maintenance that apply not only to critical systems, but to software systems in general. These include testing (the primary component of validation), version control and continuous integration, issue tracking and code review, release management, documentation practices, and process automation. This set of practices is not mean to be exhaustive. For example, we do not discuss particular code standard choices or the relative merits of various Agile development methods, as these are primarily matters of development team preference and have little bearing on the reliability of the resulting software. Instead, we focus on practices that are directly relevant to building and maintaining critical systems and generating evidence of their correctness.

7.3.1 Testing

Software testing practices are a key component of any software engineering methodology. Even when parts of a system are formally verified, testing can provide additional assurance that the system satisfies its requirements, behaves as expected for particular inputs, works correctly in diverse environments, and has sufficient performance.

Testing serves the key functions of uncovering flaws quickly and ensuring that previously-fixed flaws do not recur in later software versions. It is impossible for testing to reveal all possible flaws in any realistic program—the number of possible inputs for any such system is so large that an exhaustive test would effectively run forever—but tests that successfully reveal and prevent recurrences of some flaws provide some evidence for a system's correctness.

Verification and testing should not be viewed as opposing alternatives, but rather as complementary techniques that together provide assurance in the developed software. It is not feasible to exhaustively test every possible input and every possible path through any non-trivial program, while verification offers guarantees over all possible inputs. However, verification usually cannot scale to provide those guarantees for entire systems; verification tools must sometimes make simplifying assumptions about the environment in which software runs or reason about a simplified model of the actual system. Since testing can exercise the real system in a real environment, it can uncover flaws that are beyond the scope or capability of verification.

Different testing practices provide complementary types of assurance; no single testing practice is sufficient. Instead, multiple types of testing should be used in combination on every project.

Unit Testing

Unit testing exercises the smallest components (the "units") of a program that are feasible to test. The granularity of unit tests varies by programming language, but typically unit tests are small enough to test the implementation within a single module, class, or other per-file abstraction.

Unit tests are usually written to reflect the specification of the unit under test. For example, a unit that implements a specification of addition might have unit tests that check the associativity and commutativity of the implementation. Most software specifications are too abstract to translate directly into unit tests so, for a property like associativity, developers must choose particular concrete values to test. Unfortunately, developers might not choose the particular values that would expose a flaw; when they fail to do so, the test suite will succeed despite the presence of that flaw.

Developer intuition and understanding of the implementation increases the likelihood that unit tests will exercise code containing a flaw, but tests still cannot be exhaustive. Code coverage, the percentage of lines of code exercised by a given test suite, is often used to measure the effectiveness of unit tests. In practice, high code coverage percentages have not been shown to necessarily uncover more flaws [110], however more sophisticated coverage measures such as branch coverage are more promising [89].

Randomized and Fuzz Testing

Manually-written tests can only exercise a small fraction of the potential inputs to a program. When test cases can be generated randomly, it is much cheaper to produce a large number of test cases that can explore a larger fraction of potential inputs.

With *randomized testing*, developers specify a means for generating the inputs to a test, and provide a function or "oracle" for evaluating whether the test succeeded with those inputs. These test generators can more closely mirror specifications than tests that use concrete values, as they can instead make assertions about all possible values.

For example, a unit test for an addition implementation might by hand assert that $0+0=0$, $1+0=1$, and $2^{32}+0=2^{32}$, but a random test could assert that for all integers x, $x+0=x$. Unless the range of the input is very small, a random test will not provide an exhaustive proof of the property it expresses. It can, however, easily provide orders of magnitude more test cases than hand-written tests, and will usually produce test cases that a developer might not think to add intuitively. When a higher level of assurance is required, the specification of a random test often translates directly to a logical formula usable by formal techniques, easing the transition to a verified system [186].

In the addition example above, it is straightforward to generate random integers and check whether the results are correct. However, randomized testing is situational, since an oracle is not always straightforward to develop, and even the random input generator can be quite complicated for complex input types. For example, random testing has successfully been used to find flaws in C compilers. The development of the C test program generator has itself been the subject of extensive research [207], and the oracles used are primarily other C compilers.

Randomized testing is difficult for programs with complex input types and no readily-available oracles. However, a closely related technique called *fuzz testing* is useful for such programs. A fuzz testing tool generates random input, perhaps guided by initial suggestions; it then passes that input to the program under test and observes the result. If the program crashes or violates any of its internal assertions, the fuzz tester reports the result and the input that caused it. If the program runs successfully, the fuzz tester tries another random input. This process is repeated continuously until it is stopped or, in some cases, until the fuzz tester determines that it has fully exercised the program.

Fuzz testers use a variety of techniques to generate their test input, including pure randomness, genetic algorithms (mutating previous inputs to generate new ones) [66], and symbolic execution with constraint solving (calculating new inputs to avoid repeating execution paths that have already been tested) [92]. Fuzz testing has exposed many critical security issues in widely-used software, both open-source and proprietary.

Model-Based Testing

Another method for automatically generating test cases is *model-based testing*, where formal models of the program's behavior can be used as test oracles, as guides for choosing test data, or both.

Formal models of program behavior can be used as test oracles by compiling them into the software as *runtime assertion checks*. For example, assume the specification for some function f guarantees that some condition x is true when it returns. If x is ever false when f returns, f does not satisfy its specification. Any good set of unit tests for f should detect this divergence, and it would certainly be straightforward to write (by hand) a test oracle that checked the value of x after executing f. However, in many cases f's specification can be *automatically* transformed into a set of runtime assertions, such that (in this case) an error is always raised if x is false when f returns. Compilers that perform such transformations are available for several programming languages and corresponding specification languages; some widely-used examples are Java and the Java Modeling Language (JML), C# and Code Contracts, C and the Executable ANSI/ISO C Specification Language (E-ACSL), and Eiffel (with its integrated specification language).

The set of runtime assertion checks derived from a formal specification of a module effectively becomes a test oracle for that module; in general, more precise specifications lead to better test oracles. For a function f with such an oracle, each possible input to f defines a test, and each test is run by simply calling f with that input. A test passes if all the runtime assertion checks pass and fails if any runtime assertion check fails. If the specification of f restricts the set of possible inputs, tests that supply invalid input are effectively meaningless and their output is ignored.

Of course, it is impractical to call f with every possible input; however, strategies such as randomized input data generation (discussed above) and "interesting" input data generation (for example, ensuring that all boundary conditions are tested for data types with boundaries, such as numeric types) can be used to cover the functionality of f with a reasonable number of tests. Multiple test frameworks use these techniques to automatically generate and run model-based tests.

Formal models of program behavior can also be used as guides for choosing test data; for example, some tools use constraint solving techniques on programs and their specifications to ensure that test data satisfies input constraints on the functions under test. Techniques such as symbolic execution—analysis of the program to determine what inputs cause what execution paths to be taken—can also be used to choose test data, with the goal of achieving maximal coverage using a minimal set of test cases.

Regression Testing

In addition to unit and integration tests written to reflect the specifications of a system, new tests should be added whenever a defect is uncovered and fixed. Defects tend to recur in software for a variety of reasons. The existence of the initial defect may imply that there is some subtlety to that particular code, raising the baseline likelihood of defects. The fix applied to the code may have only fixed the defect under the limited set of conditions that were observed at the time, for example in a bug report. The fix may also have depended on assumptions about code elsewhere in the project, and the defect might recur once those assumptions change.

Running a *regression test* for every defect in the project's history assures us that those defects are not present in the current software artifacts. However, for a long project, the weight of that history can make the regression suite unfeasibly large. Many longer-term projects therefore split their regression tests into multiple suites: a small, quick suite to run before each code commit, a larger suite to run every night, and sometimes a full suite that runs over weekends or before major project milestones. Since a goal of testing is to uncover defects as quickly as possible, running tests less frequently is a tradeoff, and prioritizing and minimizing the cost of testing is an area of active research [208].

Integration Testing

While unit testing and regression testing find defects in individual modules, *integration testing* finds defects in the system as a whole. This type of testing can expose flaws in the way multiple modules interact, measure performance of the integrated system, reveal environmental (e.g., configuration, operating system, network) dependencies, and simulate the overall experience of the system's users.

The most basic integration test is simply to check that the complete system can be successfully built. Once built, integration tests exercise substantial functionality across multiple modules, often simulating the actions performed by a user during an interaction with the system and checking for expected outcomes. In this sense, integration tests are frequently the first line of validation applied to a system.

For example, in an E2E-VIV system, one integration test might load a ballot, make selections, change selections, and then cast the ballot. Another integration test might follow the same steps, but then spoil the ballot and repeat with a new ballot. While each of these individual steps might concern only a single module, the combination of steps helps expose potential problems with module interactions.

In addition to simulating overall functionality, integration tests test the suitability of the software in its intended operating environments. This is critical for systems that are intended to work with multiple operating systems, with or without a network, or with specialized hardware peripherals like a ballot marking device. Making the environmental assumptions explicit in integration tests also helps prevent defects from arising due to unstated assumptions. For example, a developer may inadvertently write software that depends on features that are unavailable in the deployed environment. It is counterproductive and often infeasible to develop in the deployed environment, which may not even be capable of running development tools; therefore, the integration tests must accurately recreate that environment.

7.3.2 Version Control

A *version control system* (VCS) manages changes between the versions of a project as it evolves during the course of development. Revision control is the preferred way to share software artifacts across a team, but all software projects, even those developed by teams as small as one person should use a VCS.

In general, a developer uses the VCS first to "check out" the files comprising a project into her "working copy". Then, after making changes to those files, the developer "checks in" or "commits" the changes to the VCS. After committing, those changes are available for other developers to integrate into their working copies.

When different sets of changes have been made by multiple developers, the VCS can merge those changes either automatically or by using developer input to ensure the project remains consistent. The ability to merge changes is critical for teams of developers who work concurrently on a single project, and is the reason that file sharing tools like Dropbox or Google Drive are an inadequate substitute for a VCS.

Version control is particularly important for projects, such as critical systems, where the development process must be auditable. An entry to a project log is created every time a developer commits their work. Any file in the project can be inspected to show its provenance, even down to the level of which line was committed by which developer on what date. Some VCS tools also allow for commits to be cryptographically signed, offering assurance that, for example, the changes have been audited by a trusted authority before being integrated into the project [40].

Moving from simple file storage to a VCS is a tremendous improvement for development, but poor use of a VCS can negate many of the potential benefits. For example, the log built from the commits of developers is of less use to auditors if the changes in each commit are not clearly associated with a particular new feature or bug fix. Likewise, if developers commit changes that cause the system to function incorrectly, other developers' productivity suffers and it becomes more difficult later to isolate the commit that introduced a bug. Each VCS supports multiple workflow practices that should be adopted in order to limit these problems and get the most benefit from VCS use [21, 159].

7.3.3 Continuous Integration

The expense of fixing software defects increases over time as other parts of the software evolve around those defects. When a defect is new, developers have not had a chance to write other code that depends on the defective code. Once such dependencies exist, a fix for a single defect can have consequences that ripple outward across the entire project, making the fix much more expensive. The cheapest way to fix a defect, then, is to discover it as quickly as possible.

Continuous integration (CI) facilitates this by discovering defects as part of a regular, automatic process that is not dependent on due diligence of individual developers. CI interleaves the quality control process into the development process, rather than leaving it as a separate phase for the end of a project after development has finished.

CI tools automatically build and test the latest version of the software in the VCS system on a regular basis, such as every night or after every VCS commit. Because the software is built from the VCS, it is important for developers to frequently commit their work to the VCS. The VCS integration ensures the tests are always run on the canonical version of the software, and that any discovered defect can be linked to a particular version in the VCS system. Isolating the failing version focuses the efforts of developers on the set of changes introduced in that version, often saving considerable time.

CI systems substantially replace manual effort and the risk of mistakes when releasing software. Since the CI system builds the project on a nightly basis, it can post the artifacts of those builds for users and testers to quickly adopt. When an official release like "Version 1.0" is ready, developers can simply run the CI system to produce the relevant artifacts. Because the same system is responsible for both the continuous testing and validation of the system and the creation of the final release, it is less likely that the final release will have defects that would have been caught through earlier testing.

7.3.4 Configuration Management

Complex software systems may depend on many components outside their own implementations, including database servers, application servers, web servers, authentication systems, etc. They may also need to run on several different target platforms with different operating systems and capabilities, or on multiple commercial cloud infrastructures with different deployment and management processes.

Configuration management is the tracking of dependencies and deployment characteristics for a system, and is typically automated by multiple tools. A *build automation tool* handles dependency tracking for building and packaging the software, and is used both by developers and by the continuous integration system. A *deployment automation tool* handles the setup and configuration of the (already built) software for a particular execution environment, such as an operating system/platform combination or a specific set of nodes at a specific cloud provider, including the configuration of the individual machines on which the software will run.

Configuration management saves significant manual effort and greatly reduce the risks of mistakes when building and deploying software, and should be used in all projects that have external dependencies or target multiple deployment platforms.

7.3.5 Software Product Lines

Developing software that has to be changed in many ways for different target audiences has historically been a nightmare. Each different version of such a piece software is called a *variant*, and the decision points in the software or its build system that derive variants are called *variation points*. This style of software development is called *software product lines* (SPL) [51, 63].

Traditionally, one would develop such software with programming languages that have either (a) support for conditional compilation, usually in the form of C-like `#ifdefs` or Lisp-like macros, or (b) inheritance, using subtyping and sub-classing as a means of selecting variants. Using the development process to achieve the same goal typically results in a large tree of branches in a VCS, where each branch is a (mostly) separately maintained variant.

These traditional styles of software development, without additional supporting technology, have been shown to be unfit for this purpose [48, 123, 133]. The resulting software is typically an impenetrable mess that is extremely difficult to maintain.

An alternative way of developing SPLs is to specify variants by declarative means. The academic literature highlights three good solutions to this problem: a specification technique called *feature models* [122]; expressing variants using a specification language in the context of model-driven development (MDD) (e.g., Clafer [49] or the OMG's Common Variability Language (CVL) [100]); and embedded domain specific languages (EDSLs).

Product support for feature modeling and CVL is in its infancy, but is rapidly evolving as the industry demands better support for developing SPL-based software systems [29, 31].

7.3.6 Issue Tracking

During much of the software development and maintenance process, teams add features and fix bugs. An issue tracking system maintains records about each new feature, each reported bug, and other discrete development tasks from their creation to their implementation, review, testing, and integration. Issues can be organized by metadata such as assignee, project milestone, priority, and task type. Issue trackers are essentially to-do lists with additional structure, specialized to support effective software engineering.

These issues and their metadata give team members an overview of the status and health of the project. For example, the issue tracker may automatically require a code review step before an issue can be resolved. Issue trackers help teams make fewer mistakes when following best-practice software engineering workflows.

Team members can annotate issues with comments or attach supplementary documents, creating a record of design decisions and thought processes. This information is invaluable when investigating future bugs or making subsequent changes to a design, and is often lost when such discussions take place out-of-band and lose their associations with the tasks that motivated them.

Most issue trackers integrate with VCS in order to associate issues with source code changes. Combined with the design discussions captured in issue comments and attachments, this enhances the ability of the team to understand and maintain the project in the future.

When issue trackers are public they can also serve as a first point of contact for users, providing insight into the evolution of the system and a well-defined process for reporting system issues. In projects with short development timelines, it is critical to incorporate feedback into development as quickly as possible. Giving users or front-line support staff the power to create issues directly makes the feedback loop very small.

Public issue trackers also reduce duplicated effort by both users and developers. If a system has a problem, that problem will likely become apparent to multiple users; duplicate reports are less likely if users can check the issue tracker for other reports of similar problems. The development team can then triage issues by importance and urgency, discuss potential solutions, assign developers to implement those solutions, and finally make sure the problems are resolved and notify the users who originally reported the problems.

7.3.7 Code Review

Code review practices involve examining the results of the software development process to find defects, identify potential improvements, and increase understanding of the software throughout the engineering team. Reviews are also an opportunity to ensure that organizational code style standards are met and that the code and its documentation are easily understandable. This process, like discovering defects during testing and investigating issue reports, feeds back into an interative development process to improve the quality of the final product.

Code review can be a manual process at varying levels of rigor. On the formal end, processes like Fagan inspection require a line-by-line inspection by many developers in an extended meeting and catch a high percentage of defects [78]. On the lightweight end, code review occurs implicitly during pair programming and can take place informally via a developer-led walkthrough or an email to colleagues requesting feedback. Formal inspections are more costly than informal reviews, but may be more suitable for projects that require concrete audit trails for accountability. Lightweight methods, particularly pair programming, can find similar proportions of defects for lower cost [192] and have other positive effects such as higher developer job satisfaction and improved team dynamics [53].

Automated tools complement any form of manual code review. Lightweight static analysis tools and code "lint" tools can help developers avoid common coding mistakes and adhere to organizational style standards. Very lightweight static tools can be run by individual developers before committing their work to the VCS, and longer-running analyses can be part of the continuous integration and testing process. Such analyses are not substitutes for formal verification or testing, however, as they typically are meant to discover small-scale defects and help developers avoid common pitfalls rather than to validate overall properties about the correctness of a system.

7.3.8 Release Management and Lifecycle

Release management is the transformation of a set of software artifacts into a finished product that can be used in its intended environment. Release management is primarily focused on the smooth integration of the different aspects of the project and on adhering to practices that make releases repeatable, reliable, and auditable.

Release management and VCS workflows are tightly connected. For example, in the Git Flow model [21], release management would include creating a new release branch, imposing a feature freeze (no new features, only bug fixes) on that branch, and eventually tagging that branch upon release and merging it back into the main development branch.

For a project that delivers software as a service on a web server, release management would include deploying the software to production servers. For software delivered as a binary download or CD, release management would include cryptographically signing and distributing the binary. In both of these cases, a release manager serves as the final line of quality assurance before the software is used in its intended environment, and must be fluent enough with all aspects of the project and its processes to release software only once the processes have been faithfully executed.

7.3.9 Testable Documentation

Documentation of the design, implementation, and use of a software system is a standard requirement in software engineering methodologies. However, when a system is under development and rapidly changing, documentation can lag behind and fall out of step with the latest version of the software, leading to errors and confusion.

Where possible, documentation should be machine-testable (or even machine-generated) and integrated with the VCS rather than being a set of static resources maintained independently of the software. Testable and generated documentation is far less likely to become inconsistent with the software it describes, because any such inconsistencies will be detected during testing.

The form of testable documentation varies depending on the granularity of the documentation and the underlying technologies used by the project. For example, Business Object Notation (BON) [202] can be used as analyzable documentation at the specification, design, and architecture level.

At the level of code modules and interfaces, documentation should be concretely executable like the "doctest" features available for specification languages like JML (using its `examples` pragma) and programming languages like Python and Haskell [83]. Documentation in this style contains short examples that illustrate the expected use of a system and its expected response, for example in Python:

```
"""
This is the fibonacci module. It provides the function fib which
returns the nth fibonacci number, where n >= 0.

>>> fib(0)
0
>>> fib(10)
55
>>> fib(-1)
Traceback (most recent call last):
    ...
ValueError: n must be >= 0
"""
```

Executable tests should supplement, not replace, traditional prose documentation. Since they are essentially a form of unit test, they suffer from the same limitations. They typically only exercise a handful of concrete values, and cannot test non-functional properties like expected performance or thread safety.

7.3.10 Reproducibility and Automation

A team can implement many of the techniques in this section manually. Tests can be run by hand on developers' machines, code can be sent out for review by email, issues can be tracked on a mailing list, and a release manager can build and package release artifacts by hand for each supported platform. However, each time a step in a process must be manually performed, the probability of human error increases and reproducing steps for later quality assurance and troubleshooting becomes harder.

A manual operator might skip a step or perform a step out of order, for example running the test suite before integrating the latest changes from the VCS. The operator might also introduce new steps that seem necessary and obvious, but unless recorded will make it very difficult to reproduce or audit the process in the future. Finally, manual execution of a process, even if done correctly, takes much longer than automated execution.

To prevent errors, improve reproducibility, and make development more efficient, processes should be automated as much as possible. The techniques described in this section all support automation and reproducibility or can themselves be automated to a degree, but some play key roles.

Version Control The version control system is a linchpin of automation. The versions it manages are the starting point for automated and reproducible continuous integration, testing, and software releases. The VCS can itself trigger automated processes; for example, it could trigger a run of an automated test suite after every commit.

Any automated process should be run in the context of a particular VCS version, and any artifacts produced by these processes should refer to this version. For example, if software has a built-in bug reporting feature, those reports should automatically include the version at the time the software was built. This allows engineers to easily reproduce the exact circumstances where a user discovered a defect.

Version control can only improve reproducibility when all relevant inputs to a process are managed in the VCS. For example, a software build process that depends on a configuration file in the user's home directory would not be reproducible on a different computer without that home directory.

Testing Manual testing can play an important role when evaluating a system, but any realistic system requires more tests than are feasible to perform manually. Even if the contents of a test suite are automatic, if that test suite is only run manually it will often be skipped, particularly when it takes a long time to run. Automating both the tests themselves and the running of those tests ensures that they will be run on a consistent basis, and that the results will be reproducible and traceable to a particular version.

Continuous Integration Continuous integration is another linchpin of project automation. Since CI tools are designed to automatically run on a regular basis and offer integration with the VCS, other processes are usually automated by using these tools. For example, after building the integrated software, a CI tool should run the test suite and archive the built artifacts for subsequent release management.

Configuration Management Configuration management is another area where automation is critical. Tracking the software's dependencies and handling the details of its deployment on various platforms should be left to automated build and deployment automation tools to the extent possible, and these tools should be run frequently. Typically, configuration management tools are used by the continuous integration system in addition to being used in release management.

Release Management No process has as many moving parts or cross-project concerns as release management, making manual release management extremely error-prone. The entire process, from checking out a version from the VCS to deploying the final release artifacts, should be as automatic as possible. The manual intervention should amount to simply deciding which version to release and checking before the final release that the automated process performed as expected.

Because automation is inexpensive when using continuous integration and configuration management, it is a good practice to have those tools perform parts of the release management process on a regular basis, even when software is not ready for a release. If the process of producing a release is the same as performing an ordinary nightly build and test, it is less likely that problems will arise only at the release stage when it is much more costly to address those problems.

7.4 Technology Recommendations

Here, we provide some recommendations about specific technologies that, at present, are well suited (and in some cases, *not* well suited) to performing rigorous software engineering of the type we have described. These recommendations are based on experience applying these methods over the last 15+ years, but they are not meant to be a rigid set of rules or to unconditionally exclude technologies not mentioned here. The landscape of software development languages and tools is constantly changing; new languages and tools appear, while old languages and tools disappear, are marginalized, or evolve in possibly surprising ways.

7.4.1 Domain Modeling

For domain modeling, we recommend the Business Object Notation (BON) [202] and Extended Business Object Notation (EBON. BON is both a language and a design/refinement method encompassing informal domain analysis and modeling, formal modeling, and implementation-independent high- and medium-level specification. BON has a well-defined semantics, is easy to learn and write (especially the informal models, which are effectively collections of simple English sentences), and has equally-expressive textual and graphical notations. BON was originally developed for use with the Eiffel programming language and is well-supported by the EiffelStudio tool suite; however, it can be used with other specification and implementation languages. EBON adds additional *semantic properties* to BON, allowing BON to express properties relating to domains such as concurrency, ownership, responsibility, bug tracking, literate programming, and version control.

We recommend BON over the Unified Modeling Language (UML), the de facto standard for modeling in the software industry, for several reasons. First, BON's equivalently expressive textual and graphical notations are easy to work with and manipulate. UML supports only a graphical notation, though there is also an unsupported official "Human-Usable Textual Notation" [149]; it was last updated in 2004, reflects only the version of UML that was current at the time, and is not as expressive as the graphical notation. Tool support currently exists for at least a dozen different and mutually-incompatible textual UML dialects, none of which are as expressive as the UML graphical notation and most of which have significant readability issues.

Second, BON's semantics are an integral part of the language and method and are easily understandable. By comparison, UML effectively has no semantics; the Object Constraint Language (OCL)—the specification language typically associated with UML models—is a very complex expression language, and is not an integral part of UML.

Third, BON explicitly supports (and encourages) *seamlessness* and *reversibility*. Seamlessness is the property that allows a BON model to be smoothly (and, in many cases, completely automatically) refined to lower-level specification languages, and further to executable implementations. Reversibility is the property that allows consistency to be maintained between the BON model, which is an important part of the system documentation, and the resulting implementation—when the implementation is changed, that change can be (again, often completely automatically) propagated back up to the BON model. Seamlessness and reversibility are both useful properties for ensuring that the final software product accurately reflects the original domain analysis and architecture design.

Fourth, BON supports high-level domain modeling using natural language, making it easy to communicate models not only among software developers but also with other stakeholders in the development process. The BON representation of the E2E-VIV requirements [126] consists almost entirely of simple English sentences; it is therefore far more accessible to a wide audience than an equivalent set of UML diagrams would be.

Finally, BON is *simple*. Its specification is a fraction of the size of the UML specification, and its graphical representation is significantly less complex. For example, arrows in BON diagrams only have two possible appearances (a single or double line, with a single filled arrowhead) as compared to UML's proliferation of arrow types (solid and dashed lines, filled and empty arrowheads, diamonds, and circles, and additional connection markings).

While we recommend BON based on our experience, it is not the only reasonable choice for domain analysis in a rigorous software engineering context. The Vienna Development Method (VDM) [156], Z [56], the B Method [139], and the Rigorous Approach to Industrial Software Engineering (RAISE) [166] are all well-established domain analysis methods with textual representations, well-defined semantics, and tool support.

7.4.2 Formal Specification

In addition to its use for domain modeling, BON can also be used as an architecture specification and concept specification language and we recommend its use as such. Once written, BON specifications can be refined further into architectural specification languages for expressing detailed architectural properties, source code specification languages for integration with particular implementation languages, and protocol specification languages for formalizing interactions within the system.

UML is the most commonly used tool for architecture specification, but is not well suited to rigorous software engineering because it lacks semantics. We recommend using BON for high-level architecture specification and a dedicated architecture specification language, such as the SAE Architecture Analysis and Design Language (AADL) [80] or the OMG Systems Modeling Language (SysML) [151], for low-level architecture specification. Several tools support the creation and manipulation of AADL and SysML specifications and automatic generation of code from architectural models; some tools, like OSATE2-Ocarina [154] for AADL, also support automated verification.

Programming languages for which there is an obvious best choice of source code specification language include Java (JML [130]), C# (Code Contracts [54]), and C (ACSL [6]). Some implementation languages, such as SPARK [181] (a dialect of Ada) and Eiffel, have integrated specification languages. These specification languages all have several features that we consider essential for efficient and effective use: straightforward syntax and semantics, integration with widely-used software development environments, and tool support for performing analysis and verification. In the case of JML and Eiffel, tool support is also available for automated reversible refinement from BON.

In any given project, different parts of the architecture may be implemented in different languages. For example, it might be appropriate to implement some computation-intensive parts of a design in a language like C while implementing the rest of the design in a language like Java or C#. Thus, multiple source code specification languages may be used in a single project, though we recommend that high level specifications be written in a single language to the extent possible.

There are also several language-neutral specification languages and associated development tools, such as Alloy [10], Coq [187], Event-B [2], VDM [156], and Z [56], which themselves support refinement to various implementation languages. Most have associated tools for *model-based synthesis*, the process of automatically transforming formal models into conforming executable implementations. These languages and tools can be used effectively, either by themselves or alongside other specification languages, in a rigorous software engineering process; they are especially useful for specifying, verifying, and automatically generating small, critical system components.

Finally, any of several protocol specification languages can be used to specify interaction protocols within the system. These include the High-Level Protocol Specification Language (HLSPL) used by Avispa [18], typed π-calculus as used by ProVerif [28] and CryptoVerif [60], Casper [135], EasyCrypt [71], and Scyther Protocol Description Language (SPDL) [59]. Each language is directly tied to a protocol verification tool, and each tool has its own strengths and weaknesses with respect to verifying different types of protocol; thus, the choice of tool for verifying a given protocol dictates the choice of language for specifying that protocol (or vice-versa), and it may be appropriate to use different tools for different protocols within the system.

7.4.3 Implementation Language

The choice of implementation language is clearly important, and there are many possible choices; hundreds (possibly thousands) of programming languages exist and dozens of those are actually viable, with widespread adoption, tool support, and support communities. In most cases, language choice determines programming style: for example, Eiffel imposes an pure object-oriented style while Haskell imposes a pure functional style. Language choice also determines the set of existing functionality, in the form of standard libraries accompanying the language or well-established external libraries available for use with the language, on which developers can rely while building the implementation.

Implementation languages are distinguished by different styles, different methods for error handling, different security guarantees, and different ways in which programmers can make mistakes (both minor and catastrophic). For rigorous software engineering, we generally recommend languages with strong type systems that actively prevent programmers from making any of a large class of errors. We also recommend languages that automatically handle memory allocation and deallocation, which prevents another large class of errors and many potential security issues. Finally, we recommend languages that either have good specification and verification tool support or are designed explicitly for the implementation of high-assurance software. It is certainly possible to implement reliable software in languages that do not have all these features—for example, code can be written in a safe subset of C, specified with ACSL, and verified with various tools—but it is significantly more difficult to do so.

The nine implementation languages we currently recommend for rigorous software engineering are (in alphabetical order) C (a safe subset such as C_0 [36] or the C dialect supported by the Verified Software Toolchain [200]), C# (excluding unsafe code), Eiffel, Erlang, Gallina (the executable specification language for the Coq proof assistant [187]), Haskell, Java, OCaml, and SPARK. No single implementation language is ideal for every project, and it is often appropriate to use multiple languages in the same project. For example, it would be reasonable to write the computational core of a voting system in a pure functional language like Haskell and the voter-facing user interface components in an object-oriented language like Java.

7.4.4 Static Analysis

Static analysis tools process the system's source code and specifications to provide information about the system without executing the code. There are many forms of static analysis, ranging from simple syntactic checks to full functional verification. The following set of recommendations is not exhaustive, and in particular does not discuss specific tools covering all the recommended types of static analysis for all the recommended programming languages; many useful static analysis tools that can be effectively deployed in a rigorous software engineering process are not mentioned here.

At least one static analysis tool should be used to enforce some set of code style and formatting guidelines, so that the implementation's code base is consistently readable. Examples of tools that enforce such guidelines are Checkstyle [47] for Java and StyleCop [184] for C#. The style enforcement tool(s) should be run automatically and frequently, either within each developer's environment (for example, a Checkstyle plugin can run Checkstyle every time a file is saved in any of the commonly-used Java IDEs) or as part of the version control or continuous integration processes.

Static analysis tools should also be used to detect "code smells" and other problematic aspects of the implementation. The presence of significant amounts of duplicate code in multiple locations in the implementation and the excessive use of numeric literals that should be declared as symbolic constants are examples of code smells; other problematic aspects might (depending on the implementation language) include memory leaks, buffer overflows, and concurrency problems. Static analysis tools that can detect these issues include FindBugs [81] and PMD [160] for Java, ReSharper [168] for C#, Eiffel Inspector [72], and SPARK Pro [182].

Finally, static analysis tools should be used to verify protocol specifications and source code specifications. As previously mentioned, the choice of tool for verifying protocol specifications is dictated by the choice of protocol specification language. Source code specifications can be verified with extended static checking tools, such as OpenJML [153] for Java with JML specifications, Frama-C [84] for C with ACSL specifications, and SPARK Pro. In addition, source code can be proven equivalent to a reference implementation or to a formal model using tools like the Software Analysis Workbench (SAW) [172]; this type of analysis is particularly useful when verifying the correctness of cryptographic algorithms, which are pervasive in E2E-VIV systems.

7.4.5 Dynamic Analysis

Dynamic analysis tools monitor a running system to measure aspects of its operation and detect undesirable behavior. There are many different forms of dynamic analysis and many dynamic analysis tools; like the static analysis recommendations above, the following set of recommendations is not meant to be exhaustive.

One important form of dynamic analysis that we strongly recommend using when possible is runtime assertion checking (RAC), previously mentioned in associated with model-based testing. RAC compiles runtime checks for specification conformance into the implementation, which ensures that any runtime violations of the source code specifications will be detected and reported. OpenJML for Java, Code Contracts for C#, SPARK 2014, and Eiffel all support RAC compilation with their associated specification languages.

Another form of dynamic analysis that can be useful is *specification inference*. Tools like Daikon [74], which works on Java and C# programs, can infer likely specifications (particularly invariants) that may have been missed during the specification refinement process. If these inferred specifications are valid, and are added to the source code specifications, they can assist ESC tools in verification and provide additional runtime checks; in some cases, the additional specifications can allow ESC tools to verify parts of the implementation that they otherwise could not.

Coverage analysis is another useful form of dynamic analysis, particularly when used in conjunction with automated testing. Coverage analysis tools, such as EMMA [73] for Java, OpenCover [152] for C#, and GNATcoverage [90] for SPARK, can provide information about "how much" of the code was executed during a particular run. "How much" can be expressed using several different metrics, including *statement coverage* (the percentage of the source statements actually executed), *branch coverage* (the percentage of the program branches that were taken during the execution), and *path coverage* (the percentage of possible execution paths that were taken during the execution). When used in conjunction with automated testing these coverage metrics can give a rough idea of test suite quality; for example, a test suite that does not exercise the entirety of the system by at least one metric is clearly not as good as a test suite that exercises the entirety of the system by all coverage metrics.

Dynamic analysis can also be used to detect issues related to memory (leaks, corruption), concurrency (deadlock, spinning, data races), resource allocation (unclosed sockets and files), and security (buffer overflows and other vulnerabilities). Some of these issues are already mitigated by the languages and other forms of analysis we recommend—for example, none of the languages we recommend allow programs to be vulnerable to buffer overflow attacks in the same way that traditional C and C++ programs can be—but it is useful to run dynamic analysis tools to detect the ones that are not. There are too many such tools, of too many types, to list here.

Finally, dynamic analysis can be used to *profile* the executable code, monitoring it to determine which parts are executed most frequently and to find performance bottlenecks. This allows developers to gather empirical evidence for use in comparing multiple implementation choices (e.g., what data structure variant to use for a particular part of the system's data model), rather than blindly guessing at the consequences of implementation decisions. It can also help to direct optimization efforts when tuning the system for performance late in the development process. All the languages we recommend have associated profiling tools, many of which are integrated into their respective development environments.

7.4.6 Model Checking

Model checkers attempt to determine whether a formal model of a software system fulfills some specification, such as a requirement that a concurrent system must not deadlock. If a model checker determines that a model fulfills a specification, then we can conclude that an implementation fulfills the specification if we can prove that the implementation's behavior conforms to the model. Many of the protocol verification tools mentioned previously use model checking techniques.

The most widely-used model checking languages/tools are Alloy [10], PVS [177], Spin [201], and UPPAAL [197]. Developers can write models in these languages by hand, but it is more efficient to automatically *extract* models from existing implementation code; this guarantees that properties proved about the model hold for the implementation. Model extraction is supported for several implementation languages,

Model checking requires an exhaustive search of the model's reachable states to determine whether any violate the specification. There are several ways to reduce the complexity of this search, such as by exploring multiple states with simultaneously in a symbolic fashion, but model checking remains intractable for large software systems. We therefore recommend that model checking be used sparingly, for only the most critical subsystems, and that it be used primarily for protocol verification.

7.4.7 Version Control

We recommend using either Git or Mercurial for version control. Both are current-generation distributed version control systems, supporting various models of collaboration, and both are well-supported. Both also have associated services—including GitHub [87] for Git, BitBucket [27] for Mercurial, and SourceForge [180] for either one—that provide repository hosting, issue tracking, web hosting, and wiki functionality. Git is currently the more popular of the two by a significant margin, but both are good options for new projects and the choice between them is mainly one of developer preference.

We recommend against using older, centralized version control systems such as Subversion and CVS. In general, their mechanisms for handling concurrent development and maintenance of multiple versions of software are significantly more awkward than those of the current-generation systems, and the single synchronization point inherent in their centralized architectures makes it more difficult for developers to work offline.

7.4.8 Issue Tracking

Many good issue tracking tools are available. If the version control repository is hosted by one of the well-known hosting services (GitHub, BitBucket, SourceForge), the obvious choice is to use the issue tracker integrated with that service. Otherwise, there are several proprietary and open-source choices for either cloud hosted or locally installed issue tracking.

Atlassian's JIRA [118], JetBrains's YouTrack [209], and Fog Creek Software's FogBugz [82] are all hosted issue trackers that integrate with both Git and Mercurial repositories. JIRA and YouTrack are also offered as standalone commercial products that can be installed and maintained locally. All three have roughly equivalent capabilities and the choice among them, like the choice between Git and Mercurial, is largely a matter of developer preference.

Trac [193] (along with its spinoff project Apache BloodHound [15]) and Redmine [167] are open-source issue tracking systems supporting both Git and Mercurial, which can be installed locally and used for free. They are also reasonable choices when installed and managed appropriately, though their user interfaces are generally less polished than those of the commercial options.

7.4.9 Continuous Integration and Configuration Management

Using continuous integration and configuration management tools is critical to a reliable software build process, and multiple good options are available. Specific choices of build and deployment automation tools are dependent on a project's implementation languages and deployment scenarios; we list a selection of widely-used tools here.

Jenkins [117] is a well-supported and widely-used continuous automation tool; it is open source, supports many languages and build automation tools, and integrates with many development environments, issue trackers, and version control systems.

Some generic build automation tools are available, such as the venerable GNU Make [91], but most build automation tools are developed to primarily support specific implementation languages. For example, Apache Maven [16] and Gradle [96] primarily support Java development but can be extended through plug-ins to support other implementation languages, and Cabal [188] supports Haskell development.

Several good choices exist for deployment automation. Ansible [12] and Puppet [164] enable the deployment of software across multiple machines running a variety of operating systems, and also handle configuration of the execution environments on those machines. Docker [70] takes a different approach, automating the deployment of software inside virtualized *containers*; this effectively gives each deployed application its own clean environment, preconfigured with all its dependencies, without the overhead of creating and maintaining a large set of physical or virtual machines. Unlike Ansible and Puppet, Docker only supports deployment to Linux systems.

7.4.10 Testing

Testing is an integral part of the development process, and many tools exist to automate the generation of unit tests and the execution of unit, regression and integration tests. In general we recommend that tests be run as often and as non-interactively as possible, preferably both as part of a continuous integration process and by individual developers as they implement specific parts of the system.

Test automation frameworks are essential. Many unit test automation frameworks take the same form, originated by the SUnit [185] framework for Smalltalk in the late 1990s, and are typically referred to as *xUnit* frameworks. Developers write (or generate) test cases in a special format, combine them into test suites, and execute them using a test execution program that gathers information about which tests pass, which tests fail, and how any test failures occur. The test execution program then presents this infromation to the developer in a textual or graphical format. JUnit [120] and TestNG [25] for Java, NUnit [148] for C#, and HUnit-Plus [107] for Haskell are examples of such frameworks. We strongly recommend the use of an xUnit framework for test automation, especially for complex scenario tests that cannot be automatically generated by other testing tools.

Randomized testing tools, such as the original QuickCheck [165] for Haskell and the many QuickCheck-like tools developed for other languages (most of which are named "QuickCheck for X" or "X-QuickCheck"), automatically generate random unit tests. Many of these tools are guided in their test data choices by performing constraint solving on existing source code specifications; others require manual guidance on the part of the developer. We recommend that randomized testing be used in most projects, especially for automatic generation of simple unit tests that would otherwise require significant developer effort.

Fuzz testing tools are particularly useful for exposing security issues. These tools intentionally test invalid, unexpected, and random data in an attempt to induce failures. Fuzz testing is primarily applicable to software modules that directly process external input, such as command line tools and Internet servers, and we strongly recommend performing fuzz testing on all such modules in a system. Fuzz testing tools with varying levels of speed and effectiveness are available for most languages; the most prominent example is AFLFuzz [11], which was originally designed for C but is runnable (at the cost of some efficiency) on binaries compiled from any language. AFLFuzz is extremely effective and has revealed bugs in many widely used software packages, including several critical security issues.

Since all systems implemented using rigorous software engineering techniques have at least some formal specifications, we also recommend that some form of model-based testing be used if appropriate tools are available for the implementation and specification languages. Several test frameworks use model-based testing techniques to automatically generate and run tests, including JMLUnitNG [211] for Java/JML, AutoTest [142] for Eiffel, and PEX [191] for C#.

7.4.11 Roots of Trust

All computing systems have multiple layers of abstraction; the lowest layer is the hardware on which the system runs, followed by the firmware, the operating system (which may itself have multiple layers), and finally the application software. In general, higher layers of the system must trust that lower layers are not malicious and are performing according to their specifications. The *roots of trust* in a computing system, also knoare the hardware and software components of the system that are inherently trusted.

For example, in current general-purpose computers, the boot firmware—the first code that executes when the machine is powered on—is a root of trust. If the boot firmware is secure, the system can use it to "bootstrap" a chain of trust; for example, the system can use trusted cryptographic functions to verify the integrity of software modules before loading them, and the loaded modules can then be trusted to perform other functions. However, if the boot firmware is compromised in some way, it can inject malicious code into any software that it loads, and as a result none of the system can be trusted. It is therefore critical to ensure that the boot firmware, and any other roots of trust in a system, are protected from tampering.

Currently, the only available way to ensure the integrity of a system's roots of trust is by using a piece of dedicated hardware called a Trusted Platform Module (TPM) [97]. A TPM can check the integrity of the device's boot firmware before allowing it to run, and can also provide secure storage for encryption keys to protect the contents of a machine's disk and authenticate authorized users before allowing the machine to boot. By design, the integrity of a TPM can only be compromised by a direct attack on its hardware such as physical delamination of the TPM chip or direct measurement of its radiation output. Thus, if the hardware has not been physically tampered with, the TPM and the functionality embedded within it can be considered secure and used to bootstrap trust at higher levels of abstraction.

TPM functionality is embedded into many of today's computing systems. However, the availability of TPM functionality is not enough; the layers of software above a TPM must use it properly in order to provide any assurance. Therefore, when building an E2E-VIV system, it is essential that the roots of trust of the system be explicitly enumerated and that the chain of trust for each originates in secure hardware.

7.5 Evidence-based Elections Technology

Many of the technologies discussed in this chapter produce, either directly or as a side-effect of their intended use, machine-generated and machine-checkable evidence of system correctness or security. Consequently, it seems sensible for federal and state certification standards—particularly those issued by NIST and adopted by the EAC—to at least mandate the production of such evidence, if not require the use of specific processes, methodologies, and technologies.

Unfortunately, this is not the case. Current and draft federal standards called the Voluntary Voting System Guidelines (VVSG), which describe how voting systems should be tested and certified for use in public elections, do not mandate the production of any such evidence. Moreover, the certification gauntlet that voting systems must survive, which focuses on processes and checklists, is based upon thirty-year-old thinking that has long since been dismissed both in the academic literature and by the industrial software engineering community. Researchers and developers in the security and correctness communities consider a "quality process" that produces nothing but a pile of paperwork unacceptable for developing systems that are meant to be correct and secure.

Based upon observations of the development methods practiced at existing and past vendors, mainly via public audits of voting systems, it is also clear that vendors use virtually none of the concepts, tools, and techniques discussed in this chapter. The fact that they do not use quality-centric software engineering and evidence-generating technologies is not indicative of ignorance. The fact of the matter is that there is no business pressure to do so from governments or the market, which gives existing vendors no compelling reason to change their practices in the near future.

This is a stark contrast to the state of affairs in other safety- and mission-critical systems areas: software systems for aeronautics, avionics, biomedical, nuclear energy, and spacecraft applications, to name a few, must conform to stringent development and certification standards. One might argue that election systems are at least as important, especially if they are to be used for national public elections.

Discussions with NIST have shown that they are knowledgeable about many of the concepts, tools, and technologies discussed in this chapter and are quite eager to require more evidence-generating techniques for voting system certification. This raises two obvious questions. First, if the VVSG evolves and a future version covers Internet voting systems, how should third parties measure and assess the quality of E2E-VIV systems? And second, how can non-experts, particularly election officials and the general public, interpret that evidence?

7.5.1 Measuring and Assessing Quality

The peer-reviewed software engineering literature describes many techniques for measuring and assessing the quality of a software system [32, 85, 121, 141, 145].

Several existing standards fall into the paperwork-based "quality process" category mentioned above, and should not be used in the assessment of software quality for E2E-VIV systems. These include ISO-9000 [113], CMMI [101], IEC 62304 [108], and all but the highest levels of FIPS 140-2 [79].

Several current and past international standards directly address the necessary theoretical and technical foundations for what constitutes a quality software system. These include the Common Criteria Certification [114–116], Future Airborne Capability Environment (FACE) Approach [77], the Avionics Application Standard Software Interface (ARINC) 653 [17], and the Software Considerations in Airborne Systems and Equipment Certification (DO-178B and DO-178C) [68, 69]. As such, we recommend that these existing standards should be evolved and adopted by NIST in future versions of the VVSG.

Two certification frameworks provide a reasonable starting point for future voting systems certification standards. The first is the NIST risk management framework, consisting of NIST Special Publications 800-37 and 800-53 [98, 174]), which is an excellent high-level characterization of ways to evaluate and mitigate the risks to secure system. The second is the Payment Card Industry Data Security Standard (PCI DSS) [157], issued by the Payment Card Industry Security Standards Council. The PCI DSS, like the highest levels of FIPS 140-2, helps developers carefully specify exactly the trusted code base of mission-critical systems; this is an essential exercise in the context of E2E-VIV systems.

Independent of the evolution of the VVSG, the evaluation of the quality of any E2E-VIV system—as mandated by one of the recommendations in the concluding chapter of this report—should primarily rely upon objective, machine-interpretable evidence.

7.5.2 Interpreting Evidence for Voters

Today, voters must delegate trust in the quality of their voting equipment. They delegate trust to their election officials; election officials delegate trust to certification authorities at the state or federal level; and certification authorities delegate trust to one of three (at the time of this writing) accredited test laboratories [196]. Test laboratories, which are hired by vendors to certify their products, certify a specific voting system release against a spe-

cific VVSG version. Essentially, this means that the lab manually evaluates the voting system against an enormous checklist and produces reams of documentation, but almost no evidence of the system's correctness and security that can be checked by third parties. This chain of delegated trust, which ends in a paid test laboratory that does not produce publicly checkable evidence, is clearly not sufficient for the certification of E2E-VIV systems.

In order to provide voters with verifiable evidence of an E2E-VIV system's correctness and security, we must continue to rely upon delegated trust. However, *the voters must determine the roots of that trust.*

In the E2E-VIV setting, this transparent, multi-party delegated trust is contingent upon the definition and realization of several requirements found in this report. The key requirements are that any E2E-VIV system must:

1. have a formally specified, machine-checked, E2E protocol;
2. use only publicly documented, and preferably standards-based, data file and wire protocol formats;
3. produce third-party verifiable artifacts (such as zero-knowledge proofs and independently-checkable hashchains);
4. include a comprehensive set of traceable automated functional and non-functional tests; and
5. have formal, preferably mechanically-checked, proofs of correctness and security for both the E2E protocol and its realization.

Given these evidence-generating requirements, the trust in an E2E-VIV system need not be delegated to subjectively untrusted individuals or organizations. Any organization or individual with either the requisite skills (for example, a computer science graduate student or a talented software engineer) or sufficient resources to hire a group of trusted technical experts can *objectively* evaluate the quality of an E2E-VIV system. The wide variety of individual experts, think tanks, and IT corporations across the political spectrum should make it straightforward to find enough evaluators with the necessary skills.

Moreover, given the number of well-organized and sometimes enormously well-funded entities whose focus is entirely on public elections, trusted third-party certification of E2E-VIV systems should be achievable. Certification by a small set of key organizations—perhaps the Democratic and Republican National Committees, a small set of independent top computer scientists and cryptographers involved with the Elections Verification Network or the Verified Voting Foundation, and a Political Action Committee aligned with each candidate—should be sufficient to earn the trust of virtually every American voter.

Chapter 8

Feasibility

In Chapters 1 through 4 of this report, we described the motivation for, history of, and requirements on a remote voting system that experts can approve and the public can trust. In Chapters 5 through 7, we described the necessary cryptographic, architectural, and engineering foundations, tools, and techniques necessary to design and build a system that fulfills the requirements set forth in Chapter 4. However, the fact that it seems *possible* to design and develop such a system does not mean that it is *feasible* to do so.

This chapter analyzes the question of feasibility in several areas, some of which are *technical* (correctness, security, usability, availability) and others *non-technical* (law, politics, fiscal, research, development, operational, and business). After discussing each of these areas, we summarize with an integrated feasibility analysis, focusing on the question: "Is it practical to tackle the problem of E2E-VIV at this time?"

To determine feasibility, we took multiple approaches. We examined current knowledge of these systems as discussed in peer-reviewed literature. We talked extensively with election officials, and had discussions over multiple years with both election verification activists and other experts who have decades of experience designing and developing secure, high-assurance systems.

The feasibility of E2E-VIV will ultimately be determined by those organizations with the resources to invest in E2E-VIV research and system development. Given how quickly unverifiable Internet voting systems are being deployed worldwide, such investments are time critical.

8.1 Technical Feasibility Analysis

We first examine technical feasibility. If designing and constructing a formally verified, secure E2E-VIV system is not possible, analyzing any other feasibility area is unnecessary.

Since this section focuses on technical feasibility, it refers back to the technical chapters of this report. If you are reading the non-technical version of this report, or you are uninterested in technical matters, we recommend skipping to Section 8.2.

8.1.1 Protocol

A secure and usable E2E-V protocol is an essential component of any E2E-VIV system. However, E2E-V alone does not provide many of the necessary properties for public election systems, including:

- **Coercion Resistance:** It is difficult to design a coercion-resistant E2E-V protocol where the voter votes from an untrusted computer. Coercion-resistant protocols exist where the voter votes from a trusted computer; however, these require the voter to vote multiple times, indicating each time whether or not the vote should be considered valid. Current coercion-resistant protocols pose significant usability and accessibility challenges.
- **Dispute Resolution:** An E2E-V protocol enables a voter to determine whether her vote was correctly recorded. If she discovers that it was not, the protocol does not necessarily provide her with evidence to convince a third party of the problem. It is therefore difficult for election officials, observers and the general public to determine the extent of fraud if there are multiple complaints. Existing protocols that do provide the voter with evidence of fraud require the use of paper or a second independent communication channel, as well as the use of physical election security procedures. Such procedures cannot be used for remote voting. Additionally, the use of paper or a second communication channel presents usability and accessibility challenges.
- **Resistance to Client Malware and Denial of Service:** If a voter follows all the steps for voter verification of an E2E-V protocol, she should be able to determine if her vote has been manipulated by malware. However, denial of service attacks might limit her ability to perform the verification procedure. Additionally, if the protocol does not support dispute resolution, she would not be able to provide evidence of a problem. In particular, client malware could replace her vote with another valid vote in many E2E-V designs. She would recognize this if she were able to perform voter verification, but she would not be able to prove it. The use of multiple channels or paper can provide additional protection against these types of attacks, but dilutes the usability and accessibility of the system.
- **Universal Design:** E2E-V protocols with a number of the above properties have been developed, but the use of second channels (or paper) and multiple complicated steps are difficult to achieve in an accessible system. A protocol that cannot be used properly by most voters is not necessarily secure, no matter how well it is designed. For example, if voters find it difficult to verify their votes, voter verification cannot be relied upon to contribute to the correctness of the election outcome.

Experts are divided on whether a protocol that has some of these properties would be an improvement over existing vote-by-mail systems. Further, most experts agree that it is not clear how to develop a protocol that has all of these properties. As such, this remains one of the most uncertain and challenging areas for progress on Internet voting.

8.1.2 Engineering for Correctness and Security

As previously mentioned, formal verification capabilities have advanced tremendously over the past fifteen years. Scientists were barely imagining high-assurance or formally verified operating systems and hypervisors such as seL4 [127], Mirage [144], and HaLVM [189] in the year 2000. The same is true of formally verified compilers (such as CompCert [57]), static analysis tools (such as Verasco [199]), verification tools (such as VST [200]), and verification-centric programming languages (such as Dafny [64]). Incredible advances in mechanical theorem proving, particularly for SAT, SMT, constraint solving, and logical frameworks, support all of this technology.

Design, development, and analysis of secure systems have also progressed tremendously. Powerful open source static analysis tools (such as Uno [103]), fuzzers (such as AFLFuzz [11]), and protocol specification and reasoning frameworks (such as EasyCrypt [71] and F [76]) are all publicly available and can be applied to commercial systems.

The only thing preventing the design and development of formally verified, correct and secure evidence-based systems is market pressure. Only a very small number of organizations have the necessary resources—primarily in the form of people and knowledge—to tackle such challenges, and they cannot do it without clients who provide requirements, funding, and time.

As such, if an appropriate protocol is developed, *designing and developing an E2E-VIV system is technically feasible*. We can estimate—based upon the size and complexity of the relevant protocols and subsystems—the effort necessary to build an E2E-VIV system. We can determine cost based on the estimate of effort. This analysis is provided below in Section 8.2.3.

8.1.3 Design and Engineering for Usability

Striving for security and usability often presents conflicts. Design features that offer security often decrease usability, and features that offer usability often decrease security. This problem is very apparent in early E2E-V election systems such as Helios, Prêt à Voter, and RIES. Because of this conflict, scientists are faced with a difficult question: Is it feasible to design and develop an E2E-VIV system that is secure and usable? More specifically, is it feasible to design and develop an E2E-VIV system that follows universal design principles?

Usability experts claim that universal design of election systems is reasonable and necessary. Several organizations have extensive experience in this area, such as the Center for Civic Design. Some new voting systems that are under development, such as Los Angeles County's VSAP project and Travis County's STAR-Vote system, mandate universal design.

While researchers are still studying certain aspects of usable E2E-VIV systems—particularly those of voter ritual and verifiability—usability experts agree that a universal design for an E2E-VIV system is possible in principle. The experts agree that, in order to achieve such a universal design, it is necessary to conduct a long-term, in-depth, qualitative and quantitative usability study based upon a working demonstration system.

Qualitative Experiments. In an interactive, *qualitative* experiment, a facilitator and a voter communicate using a video chat system such as Skype. The voter shares their desktop with the facilitator. The facilitator should be very familiar with the issues of E2E-VIV systems. The facilitator should also have usability and accessibility knowledge. The voter uses one of several versions of the E2E-VIV system. While using the system, the voter shares their thoughts and feelings about the experience in real-time. After the voter has finished participating in the demonstration election, the facilitator uses a script to ask the voter what they thought.

Quantitative Experiments. For a non-interactive, *quantitative* experiment, voters are solicited via social media, mailing lists, etc. to experiment with (variants of) an E2E-VIV system. Sample voters in these experiments are given ample information about what kinds of information are being collected about their behavior, so they can make a fully-informed judgement about their participation.

Various quantitative measures related to voter participation and interaction can be measured automatically, both within voters' web browsers and on the E2E-VIV server. Most of this data is similar to the analytics that any professional website collects about its users: How do voters navigate the site? Where does a voter pause for a long time to read? When does a voter ask for help? When does a voter hover over a button a long time before they decide to click it? How often do voters challenge ballots or verify their votes? How often do voters examine the bulletin board? Is there a correlation between the interactive behavior of a voter while voting and their likelihood of voting, challenging, or auditing correctly?

8.1.4 Availability

A system that is correct, secure, and usable is still not useful to voters if it is unavailable during an election. Many government websites are unreliable, especially during a distributed denial-of-service (DDoS) attack or just after a security breach.

Many companies whose businesses depend on having highly available and secure websites have effectively solved this problem. Companies like Amazon, Google, and Facebook have uptimes comparable to those necessary to run a public election, even if threatened by DDoS attacks.

The necessary network, server, and security infrastructure—and the consequent cost—to fulfill the availability demands of these companies and their customers is significant. The cost is so significant that every government that has attempted to build a facility dedicated to running Internet elections has spent many millions of dollars per election.

Today, however, there are many robust, inexpensive, public and private cloud computing platforms built to work with pre-existing infrastructure. Strong, large-scale DDoS protection services are now available. Because E2E-VIV systems do not need to run on dedicated, physically secure hardware—as long as suitable roots of trust are available—it should be possible to run highly available elections systems on existing hardware.

We believe a highly available E2E-VIV system can be deployed and maintained with current highly-available networked services. This is especially true if elections are run over a reasonable time frame (such as many days to a few weeks) and are built using a peer-to-peer network model.

8.1.5 Operational

Operational feasibility presents other challenges. Creating a correct, secure, usable E2E-VIV system that can be deployed with high availability is not enough. If that system is too difficult or expensive to integrate into existing election workflows, or is too complex for LEO IT staff to understand and support, it will not be used.

Software such as an Internet voting system is usually delivered for deployment as a bundle of source code with many dependencies. That stack of software must be hand-built, carefully customized, and installed on a server. Because of the complexity of Internet voting systems, the core application typically depends on dozens of other large pieces of software. These dependencies include databases, application servers, web servers, authenticatation servers, and dozens of libraries for processing configuration files, communicating over networks, and performing cryptography.

The vast majority of jurisdictions has neither the expertise nor the resources to deploy such a system. Deploying a traditionally designed and developed Internet voting system is not feasible.

Packaging and delivering complex distributed processing systems in cloud deployments, however, has become commonplace in recent years. Complex deployments by non-technical staff are now possible and widely available due to the creation of new technologies invented specifically to fulfill this need.

The key technologies that solve this problem are discussed in Chapter 7. These include continuous integration systems and configuration management tools for development and deployment. They also include cloud deployment and management technologies, such as those available from Amazon, Google, Microsoft, Heroku, and other major cloud providers.

If an E2E-VIV system is constructed using these specific technologies, then point-and-click deployment and management becomes a possibility even for LEO offices that have limited IT resources. In this technical setting, the operational aspects of E2E-VIV are feasible.

8.2 Non-Technical Feasibility Analysis

Non-technical feasibility of E2E-VIV systems will be primarily decided in the realms of law, politics, finance, public perception, and business interests. In particular, if any of these sectors is fundamentally opposed to any of the key features of E2E-VIV systems, then deployment of E2E-VIV systems is infeasible. The examples below are based upon discussions within the election integrity community, media reporting about Internet voting, and and reflections upon past activities within legislatures worldwide.

8.2.1 Law

In the U.S., legislators must change the legal framework of elections in nearly every jurisdiction that wishes to use Internet voting. Historically, legislatures are comfortable with introducing Internet voting trials, particularly for UOCAVA voters. Providing technology that helps disabled voters is commonplace.

However, legislatures often permit or mandate the use of new election technologies with little restriction on their form, substance, and impact. This creates serious problems. Legalizing Internet voting without mandating end-to-end verifiability, or permitting large-scale Internet voting without first evaluating the impact and success of E2E-V in polling places and in small-scale Internet voting trials, could have disastrous consequences.

Based upon historical evidence, a gradual evolution of state and local election law—particularly facilitated by the local nature of elections—seems feasible in a 5–10 year time frame.

A subsequent research and development phase of this project should include concrete legal recommendations for state and local legislators. These recommendations must ensure that the legal framework for Internet voting deployment is rational, evidence-based, and legally mandates the requirements set forth in this report.

8.2.2 Politics

In the main, politicians and elected officials want to be perceived as forward-thinking and modern. Thus, it is not uncommon for those running for office to support new election technologies such as Internet voting. On the other hand, political parties and powerful political special interest groups are motivated by other factors.

The hypothetical implications of widespread trustworthy use of E2E-VIV—particularly the possibility of increased broad-spectrum voter participation—are potentially at odds with the agendas of some political actors.

As a result, the political feasibility of E2E-VIV is an open question. Only time will tell.

8.2.3 Fiscal

The cost of developing and deploying previous non-E2E Internet voting systems is often not part of the public record. Evidence indicates that the cost of each voting system deployed in the U.S. (SERVE), The Netherlands (KOA and RIES), Norway (with Scytl), Estonia, France, Switzerland, and Australia (iVote in New South Wales and vVote in Victoria) ranges from approximately one and a half million to tens of millions of dollars. It is reasonable to expect that creating an E2E-VIV system as stipulated in this report would cost several million dollars.

Given the continuous flow of investment into elections, and the comparatively similar development cost of an E2E-VIV system, it could be considered fiscally feasible to invest in this type of innovation. Just over 3 billion dollars have been spent on elections via HAVA, the cost of non-E2E voting machines from traditional vendors is several thousand dollars per machine, and the average cost per vote in today's elections, depending upon the jurisdiction, ranges from $2 to $10 per vote.

With current election costs, an open source E2E-VIV system—even if licensed and supported at reasonable costs by commercial organizations—will likely be extremely cost-effective in the medium-to-long term. Unfortunately, federal and state funding for election technology is currently difficult to find. The U.S. Congress has no interest in expanding budgets for elections. The debate around the elimination of the Election Assistance Commission, whose yearly budget is only just over ten million dollars, illustrates this fact. States and local municipalities have tight budgets. We cannot expect any single state or municipality to fund future phases of the E2E-VIV project.

The fiscal feasibility of E2E-VIV depends, then, on non-governmental enetities that have both financial resources and an interest in the speculative impact of widely available E2E-VIV. These include non-profit foundations, wealthy individuals, and existing and new vendors that are willing to invest millions of dollars into the research and development of E2E-VIV.

8.2.4 Integration

Regarding the operational issues discussed in Section 8.1.5, an important question is whether jurisdictions' IT staff or their contractors (see Section 8.2.5, below) will be capable of integrating an E2E-VIV system into their existing technical and election workflows.

Existing Internet voting products have serious integration challenges because of their own proprietary designs and the many proprietary data formats and protocols in Election Management Systems. This situation is, in part, the motivation for the interoperability requirements of Section 4.1.8, which state that open protocols and data standards must be used and respected in any E2E-VIV system.

This idea is further strengthened by the rapid progress of the IEEE 1622 working group that focuses on standardizing election data formats and protocols [109]. Existing and new vendors are planning to revise their products to conform to the IEEE standards, especially since it is likely that a future VVSG revision will mandate their use.

For this reason, integrating E2E-VIV systems with existing local and state elections systems seems feasible.

8.2.5 Business

Election officials have historically been reluctant to develop their own technology or to rely upon technologies that do not have a significant commercial support business infrastructure. Any new IT system requires support services to cover system evolution, support, maintenance, integration, and training.

A healthy systems market will sustain the growth of an IT support infrastructure and surrounding services, including value-added resellers, system integrators, and consultants. A distribution channel and value-added support infrastructure is necessary for E2E-VIV systems to be widely accepted and deployed.

Whether such a market will develop and, if so, whether it will be a competitive market is a critical question. Given recent shifts in the elections marketplace, especially with the entrance of a new generation of vendors and technologists, we believe it is feasible that a healthy business ecosystem will emerge over the next decade.

8.2.6 Public Acceptance

The final and most crucial feasibility question for E2E-VIV is that of public acceptance. Independent of any technological or political decision, if voters do not trust the election system, they will not trust their elected leaders or their democracy.

Our initial usability study, as well as case studies in the U.S. and elsewhere, shows that the general public usually welcomes new election technology. In general, voters believe that election officials know what they are doing and that if they have chosen to deploy a new election technology, the technology must be a good one.

Trust, however, can easily break. Much of the public currently distrusts IT systems that are responsible for citizen data or services, especially since security failures of government systems and those of private companies are often in the news.

Government agencies responsible for these systems have an even harder time earning and maintaining trust. Distrust is prevalent both in government employees that must use government systems and in citizens whose private information is stored in government systems.

Earning and maintaining public trust in E2E-VIV systems will require an extraordinary amount of transparency and strategy, and will take a long time.

Because E2E-VIV systems' correctness and security rely upon deep mathematical and computer science foundations, very few citizens can directly understand them and come to trust them by reading the literature for themselves. Non-experts will always have to trust in the work of experts who develop E2E-VIV systems. The feasibility of public acceptance therefore depends on the trustworthiness of those experts and the evidence that they, and the E2E-VIV system itself, can produce.

8.3 Integrated Feasibility Analysis

This chapter's analysis provides us with the necessary components to evaluate the overall feasibility of building and deploying a practical E2E-VIV system for U.S. elections.

All technical aspects—engineering for correctness and security, design and engineering for usability, availability, operational—are feasible, though difficult.

Feasibility of the non-technical aspects ranges from unknown to entirely possible. With respect to law, feasibility is contingent upon legislators, election officials, and social pressure from voters. The financial, research, integration, and business aspects are relatively straightforward, and thus feasible.

The most important open question relates to the main challenge of any modern IT system: how do people and software relate?

The politics and public acceptance of Internet voting are open questions. We believe that only the disciplined, transparent, scientific, and practical pursuit of E2E-VIV can convince the public that E2E-VIV systems deserve their trust.

In summary, it is feasible to pursue future phases of this project. We discuss our final recommendations in our concluding chapter.

Chapter 9

Conclusion

There is tremendous pressure to build Internet voting systems and use them in public elections. Researchers, developers, and election officials must take the time to understand the requirements for secure and trustworthy elections so that they can evaluate systems—both good and bad—and make well-informed decisions. The use of flawed election systems in public elections can result in significant and irrevocable harm.

This report presents the most complete set of requirements to date that must be satisfied by any Internet voting system for public elections. This set of requirements, described in Chapter 4 and published in complete detail in a separate document [126], is useful to several audiences:

- *legislators* and their staffs who may craft laws that relate to remote elections, particularly Internet elections and elections for overseas, military, and disabled voters;
- *election officials* who may specify, evaluate, or purchase Internet voting products or services;
- *activists* who wish to better understand, and advocate for, E2E-VIV systems;
- *standards bodies* that may standardize various classes of Internet voting technologies, and specify the level of rigor for certifying Internet voting systems;
- *testing organizations* that may test Internet voting systems for compliance with technical standards;
- *researchers and engineers* who may continue working toward viable E2E-VIV systems.

This report also contains additional information useful to a subset of these audiences, including:

- a basis for developing the cryptographic foundations with which to evaluate and compare various E2E protocols and E2E-VIV systems (Chapter 5),
- an analysis of the architecture space of E2E-VIV systems (Chapter 6),
- precise recommendations on the state of the art for rigorous engineering of E2E-V systems (Chapter 7),
- a framing for an ongoing discussion about the feasibility of designing, constructing, certifying, legalizing, and deploying E2E-VIV systems (Chapter 8), and
- a reflection upon the outstanding issues that must be addressed in future stages of E2E-VIV development, including political, legal, research, and engineering challenges (Section 9.2).

Following are recommendations and possible next steps.

9.1 Recommendations

The E2E-VIV Project team does not assert that Internet voting *must* be pursued, nor does it assert that Internet voting must *never* be pursued. There is no consensus among the team members for either of these positions. With this understanding, the project team recommends the following.

Recommendation: E2E-V. **Any public elections conducted over the Internet must be end-to-end verifiable.**

The use of Internet voting systems without end-to-end verifiability—including all Internet voting systems that jurisdictions are experimenting with and using at the time of this writing—is irresponsible. Any voting systems used to conduct public elections over the Internet must be E2E-VIV systems.

Recommendation: SUPERVISED FIRST. **No Internet voting system of any kind should be used for public elections before end-to-end verifiable in-person voting systems have been widely deployed and experience has been gained from their use.**

It is critical to gain experience with E2E-V in the simpler in-person setting before attempting to deploy it in the vastly more complex Internet setting. Using E2E-V for in-person elections will also improve the integrity of existing in-person voting systems.

If election officials and election system vendors ignore these first two recommendations, we expect that deficient, unverifiable Internet voting systems will be widely used within ten years. Vendors will claim to have solved the security problems, and eager officials will believe these claims. Elections may be altered with no public awareness. If election officials manage to find evidence left by a careless attacker after altering an election, the damage will have already been done.

In making these first two recommendations, we realize that we have created a difficult path to follow. We assert that no attempt at Internet voting should deviate from this path. Building an in-person E2E-V system is no small task. Building an E2E-VIV system that satisfies the requirements in this report is even more ambitious; it may even be impossible. However, if it is possible, the resulting system will be far better than the vulnerable Internet voting alternatives.

Recommendation: HIGH ASSURANCE. **End-to-end verifiable systems must be designed, constructed, verified, certified, operated, and supported as high-assurance systems according to the most rigorous engineering requirements of mission- and safety-critical systems.**

A software independent voting system does not rely on high-assurance software to detect errors in the election outcome. However, high-assurance software engineering tools and techniques can make such errors much less likely to occur, and can also reduce the risk of the following problems:

1. **Privacy violations.** While E2E-V systems can identify and mitigate issues of election integrity, they cannot do the same for privacy issues. A poorly-implemented E2E-V system will allow observers to detect certain issues with the election (such as votes not being counted correctly), but not to detect when voter identification details are stolen from an insufficiently secured server.
2. **Programming errors.** A low-quality E2E-V system is far more likely than a high-quality one to have software engineering flaws in design, functionality, security or other areas that trigger failures in verification. These will increase the burden on election administrators to deal with partial failures. This could also significantly impact the voters' trust in the election process, as well as the election administrators, apparatus, and outcome.
3. **Security issues.** Low-quality implementations of any type of software system are extremely difficult—and often impossible—to secure in the presence of insider or outsider attack. Security is not a band-aid to apply to a poorly-implemented system; it can only be achieved through a combination of rigorous process, method, design, implementation, validation, verification, deployment, and operation.

High-assurance software engineering is the only reasonable way to attempt to implement an E2E-V system that is correct, secure, and does not have enormous fiscal and trust implications for election officials after deployment. A less rigorous development approach will almost certainly lead to costly defects.

Recommendation: UNIVERSAL DESIGN. **E2E-VIV systems must be usable and accessible.**

It is not feasible to make voting easy for voters with the most extreme disabilities. However, it is essential that we at least serve voters who have challenges in vision, hearing, comprehension, or motion yet can still use some kind of computing device. We should use a qualitative and quantitative testing-based experimentation platform to assess usability and accessibility, and follow best practices [4], recommendations [173, 203, 204], and standards [7] in accessible UI design and implementation. By doing so, we will be able to service nearly every overseas and military voter and, in the long term, the more than 84 million disabled voters in the U.S. [195].

We must also look to, and learn from, the AnywhereBallot and EZ Ballot experiments [14, 131] of the Accessible Voting Technology Initiative [3]. We must engage with the researchers and attendees at the annual California State University, Northridge International Technology and Persons with Disabilities Conference [111]. Only through direct engagement with voters, both abled and disabled, can we have any hope of understanding how to develop a usable, accessible E2E-VIV system.

Recommendation: MOVE FORWARD. **Many challenges remain in building a usable, reliable, and secure E2E-VIV system. They must be overcome before using Internet voting in public elections. Research and development efforts toward overcoming those challenges should continue.**

Building a usable, reliable, and secure Internet voting system may be impossible. Solving the remaining challenges, however, would have enormous impact on the world. Continued research and development efforts must be conducted transparently, with all results and artifacts open to peer review. Internet voting systems, including E2E-VIV systems, must not be deployed in public elections before all the key security problems are resolved.

9.2 Next Steps

To carry out these recommendations, legislators, researchers, and engineers face several challenges.

9.2.1 Political and Legal Challenges

The greatest concern voiced by election verification scientists, election integrity advocates, and E2E-V researchers is that legislators will mandate the experimentation with—or use of—Internet voting before a correct, secure, open, usable, accessible E2E-VIV system exists. Using the current untested and unverified systems opens the door to wholesale election manipulation or failure. Aggressive early adoption of election technology must be tempered by a clear understanding that voters' trust in their elections is hard-won and easily lost.

Scientists and election integrity advocates are also very concerned that well-meaning legislators and election officials will push to deploy Internet voting systems too early and too quickly, based on misleading information from prospective vendors and other advocates that is not balanced with *independent* advice from cybersecurity experts. They may also misunderstand how the security risks grow with the scale and significance of the election, and how these risks change over time as the threat environment changes. Such misjudgments may sometimes induce legistators and election officials to weigh political goals more highly than the security risks.

The political and legal challenges—and related opportunities—should focus on how to legislate the evidence-based measured introduction of new elections technologies. This includes Internet voting. *Defining an appropriate pace, milestones, and success criteria for the introduction of E2E-VIV systems must be a primary focus of any next phase of this project.*

9.2.2 Research and Engineering Challenges

Although E2E-V is necessary for a viable Internet voting system, use of E2E-V does not ensure that an Internet voting system is free from vulnerabilities. Also, the definition of an E2E-VIV protocol for U.S. elections is very challenging. In particular, the research community must determine how to address five key challenges:

- how to handle large-scale dispute resolution;
- how to authenticate voters for public elections;
- how to defend an E2E-VIV system against denial-of-service attacks and automated attacks that aim to disrupt large numbers of votes;
- how to make verifiability comprehensible and useful to the average voter; and
- how to avoid voter coercion and vote selling in the context of digital observation of voting and verification.

The usability facets of E2E-VIV are also challenging. The main issues with usability are:

- how to ensure usable vote privacy and vote integrity in the presence of client-side malware and
- how to ensure that verification is usable and accessible to the typical set of voters.

While some of these issues can be addressed by current technologies, further research is necessary to determine if all of these concerns can be adequately addressed, as discussed at length in the preceding chapters and as codified in our requirements. *Until researchers adequately address these challenges, Internet voting systems should not be used in public elections.*

The development and deployment of a high-assurance distributed system of the scale and import of a public E2E-VIV election system has never been attempted. It involves considerable engineering challenges in addition to the fundamental research challenges already mentioned. *If the research issues can be solved, current best practices for building high-assurance distributed systems should be sufficient to address the engineering issues.*

Coda

Many people believe that Internet voting will increase voter participation, help with voter decision-making and engagement, provide equal opportunity for voters with disabilities, and decrease election costs.

Proponents of E2E-V election systems hope that their adoption will prevent corrupt election officials and governments from manipulating election outcomes, and will truly capture the voice of the people and increase confidence and trust in government.

Trustworthy democracy is a worthwhile goal, and we should strive to achieve it. The only responsible way to make progress is to continue peer-reviewed research and experimentation.

Appendix A

Expert Statements

The following expert statements are all included unedited, in their entirety.

A.1 Josh Beneloh

The Viability of Responsible Internet Voting

Remote voting[1] entails significant risks above and beyond those of in-person poll-site voting. Included among these are risks to integrity – as remotely-cast ballots may pass through numerous hands without independent observation – and risks to privacy – as voting takes place without the benefit of publicly-enforced voter isolation.

Internet voting substantially exacerbates the risks of remote voting by making it possible for small problems to be magnified and replicated on a large scale. Careless or malicious errors, intrusive malware, and unforeseen omissions – all of which can be caused by individuals or very small groups – can cause very large numbers of votes to be changed and the privacy of large numbers of voters to be compromised.

The technology known as end-to-end (E2E) verifiability allows individual voters to verify that their intended votes have been properly recorded and that all recorded votes have been properly counted. When applied to in-person voting, E2E-verifiability provides new assurances to voters by allowing them to check for themselves that the results of an election are correct. When applied to Internet voting, E2E-verifiability mitigates some of the risks described above – but does not eliminate them: voters are able to check that their ballots are properly recorded and counted, but malware can still compromise privacy, prevent voters from casting their ballots, and otherwise hinder voters.

Although E2E-verifiable election technologies have existed for more than thirty years, their use has thus far been limited to small demonstration systems and private elections for student governments, professional societies, and the like. E2E-verifiable elections produce new challenges and complications for implementers and administrators. They represent a new and different paradigm for elections – substantially replacing the notion of verification of election equipment with that of verification of the integrity of individual elections. As such, it is important to act deliberately and gain experience with E2E-verifiability in more manageable environments before attempting to deploy E2E-verifiable elections in their most challenging environment: the Internet.

These realities lead us to two principal conclusions.

- Public elections should not be conducted over the Internet using systems that are not end-to-end verifiable.
- End-to-end verifiable Internet voting systems should not be used before end-to-end verifiable poll-site voting systems have been widely-deployed and experience has been gained from their use.

[1] Remote voting is defined here as voting without the benefit of the public monitoring that takes place in a traditional poll site.

The second of these two principles is also necessitated by the fact that an E2E-verifiable election must have a tally to verify, and if an E2E-verifiable system is used only for remote voters, then the votes of these remote voters must be separately tallied and reported. Few jurisdictions are willing to segregate and report the tallies of local and remote voters separately.

We take no position here as to whether the integrity benefits of E2E-verifiability and the privacy benefits it makes possible outweigh the risks of remotely-executed large-scale corruption of an Internet-based election, but we are agreed upon the conclusions that "naked" Internet voting is dangerous and irresponsible and that E2E-verifiability should be deployed in the less risky and more manageable scenario of in-person poll-site voting before it is deployed in the wilds of the Internet.

A.2 David Jefferson (Candice Hoke, Ronald L. Rivest, Barbara Simons, Philip Stark, and Vanessa Teague, Concurring)

A.2.1 Election security is national security

In a democracy election security is a key part of national security. The very legitimacy of government depends on the fact, and also the public perception, that the outcome of every election fairly represents the will of the people. We must be assured that no part of the voting process is unfairly manipulated to produce a different outcome, that all and only those who are eligible to vote have the opportunity to do so, that no one votes more than once, and that the privacy of the ballot is not compromised.

In this paper we demonstrate that while end-to-end verifiable (E2E-V) voting systems have a great deal to offer, but when they are embedded in an Internet voting context (E2E-VIV) they still do not provide sufficient security to prevent remote attackers from silently modifying votes and changing the outcome of elections undetectably, or from disrupting the election and disenfranchising a large number of voters. Fundamental security problems remain with E2E-VIV systems for which there are no practical solutions in sight and that will not be resolved in the foreseeable future.

A.2.2 Verifiability

Election security depends on *verifiability*. After an election is closed there must remain enough evidence for anyone who doubts the results to re-examine and rationally determine whether the winners were called correctly. We need to verify that only eligible voters voted, that no votes were lost, or duplicated, or modified, that no phony votes were inserted, that every eligible voter who tried to vote was able to do so, and that all the votes were counted correctly, once and only once. That evidence trail has to be *end to end*, spanning the entire data path through which the ballot data travels, from the voters' heads to the final result. We must be able to reconstruct the vote totals (or statistically audit them) from the original unmodified ballots or copies of those ballots that are provably identical to the originals as intended by the voter.

The traditional approaches to verifiability are all based on physically secured and indelible paper ballots (or paper cast vote records) that can be recounted or audited by humans without having to trust any software or complex machines. The goal of end-to-end verifiable (E2E-V) systems is to use cryptographic protocols to achieve for all-electronic voting systems the same (or higher) level of confidence in the election outcome for electronic voting systems as is achievable with paper-based systems. The goal of end-to-end verifiable *Internet voting* (E2E-VIV) systems is the same, but specifically extended to the much more difficult security context of remote voting from private platforms and devices over the public Internet.

Some approaches to end-to-end "verification" at first sound great, but fall far short. It is not sufficient, for example, to provide a feature whereby each voter can verify that her own ballot was correctly transmitted to the server over the Internet. First, it is difficult to provide verification capability without also making it possible for the voter to prove to a third party how she voted, thereby enabling automated vote selling and voter coercion. But even if that problem were resolved and if every voter verified that her voted ballot was properly received, it would still not be possible to demonstrate that no phony votes, unassociated with any actual voter, were inserted.

A.2.3 The power of E2E-V systems

End-to-end verifiable voting systems are a major conceptual and mathematical step forward from conventional voting systems. Through advanced cryptographic techniques these varied systems, while differing in many ways in their architecture and in the roles of insiders, share the following fundamental security properties:

a) **Integrity:** Once a voter successfully enters her ballot into the E2E-V system it cannot be undetectably lost or modified in any way, even in the presence of bugs or malicious logic.

b) **Privacy:** Once a ballot enters the E2E-V system it is encrypted, so that there is no way that the privacy of the ballot to be violated subsequently.

c) **Counting accuracy:** The ballots cannot be miscounted without that fact being detectable.

d) **Universal public verifiability:** The systems output and publish sufficient verification data so that *anyone* can verify that no ballots were lost or modified and that the votes were properly counted. The verification data provides essentially a *cryptographic proof* that ballot integrity was preserved and the counts are correct. Anyone is free to run a verification program over the verification data to confirm it. You don't even have to trust the official verification program—you can use one from a source you trust, or if you have the skill you can write your own.

e) **Openness and transparency:** The code for E2E-V systems is generally open source. The mathematical principles underlying the E2E-V security guarantees have been vetted by many cryptographic experts and are open and public. And the specifications for proof checkers are also publically documented so that mutually suspicious political groups can hire their own experts whose independent election verification programs, if correct, must agree.

These powerful E2E-V security properties are not shared by any traditional voting system. In a precinct voting context they make E2E-V systems essentially invulnerable to ordinary software bugs, to malicious code inside (but not outside) the voting system, to transient hardware faults, and to most kinds of insider fraud (at least without a large conspiracy). Such failures will at least be detected because any lost or modified ballots and any miscounts will be flagged whenever anyone runs a verification program. This is in strong contrast to other forms of purely electronic voting systems which are unverifiable, and in which bugs or malicious logic can cause errors that are totally undetectable. The wrong people may wind up taking office without anyone knowing that the election results were incorrect.

For these reasons it is appropriate to consider E2E-V systems in any electronic voting situation. E2E-V adds truly powerful security guarantees, particularly in a precinct voting situation where voter authentication is done in person and where we have good reason to presume that the certified software in the voting machines is not malicious.

But as we explain in the next section, these E2E-V security guarantees *do not fully extend to Internet voting systems.* E2E-V Internet voting systems (E2E-VIV) have exploitable security holes for which there are no good solutions today and that preclude them from being suitable for use in public elections for the foreseeable future.

A.2.4 Remaining unsolved security issues with E2E-VIV systems

Once the voters' choices are safely input to a precinct-based E2E-V system many, but not all, of the security guarantees described in the last section follow directly. But that is a key qualification: these guarantees begin *once the voters' choices are safely input to the E2E-V system,* but not before. Unfortunately, when an E2E-V system is embedded into the Internet voting context as an E2E-VIV system, new security problems appear that the E2E-V guarantees do not address and that cannot, with any current technology, be fixed. The problems with E2E-VIV systems arise *before the votes even enter the system.* In this section we enumerate the remaining difficult security problems that will have to be solved definitively before we can consider deploying an E2E-VIV system.

Voter authentication

A central issue with all remote, online voting systems is voter authentication. The voting system must be able to positively identify the voter in strong a way, so that it is essentially impossible to avoid the authentication process, and impossible to fool it so that an ineligible person is allowed to vote or that someone can fraudulently impersonate another voter.

Voter authentication is not part of an E2E-VIV system, but is a separate security issue. Unfortunately it is a very difficult and complex problem that remains unresolved (in the U.S. at least) for the foreseeable future.

Strong voter authentication is required for several reasons. Any online voting system must:

- verify that potential voters are duly registered or eligible to vote in the jurisdiction they attempt to vote in;
- prevent anyone from voting more than once; and
- resist vote selling, vote coercion, and proxy voting insofar as possible in a remote voting situation.

In an online voting system it is not sufficient to use the kinds of authentication commonly used in ecommerce situations, e.g. passwords, challenge-response systems based on personal information, or email-confirmations. These very weak authentication systems are more or less sufficient in commercial situations where secrecy is not so important, and where fraudulent transactions can be detected eventually and frequently be reversed or at least can be absorbed as a cost of doing business.

But in the national security context of an online election such weak authentication mechanisms will not suffice. If an attacker has the technical means to impersonate one voter, he can generally automate and amplify his methods to impersonate thousands of voters with very little additional effort. Almost every month we hear of huge data breaches at commercial or government institutions that have already allowed vast amounts of personal information on tens of millions of people to fall into the hands of criminals or foreign powers. Thus, any authentication mechanisms based on merely presenting personal information (name, address, account number, driver's license or social security number, mother's maiden name, etc.) is hopelessly compromised already, and way too weak for use in an election. Unfortunately some states have implemented such embarrassingly weak online authentication systems and have been forced to strengthen them, though they are still not sufficiently strong.

The traditional voter authentication method is based on wet ink signature matching. The voter is required, either in person at the precinct or on the envelope of a mail-in ballot, to duplicate with a new ink signature the old signature image on file from the time she registered to vote. In some states this is augmented with VoterID requirements at the polls. But there is no way to securely (unforgeably) input a wet ink signature image to a computer or mobile device and transmit it over the Internet for authentication with the ballot. Nor is there yet a way to securely and unforgeably transmit any of the usual VoterID documents.

Many people have suggested voter authentication systems based on biometrics such as fingerprints or retinal scans. For very good reasons too numerous to fully explain here none of these mechanisms is suitable for online voting. And it is important to note that while some mobile phones and tablets have fingerprint authentication devices built in, such systems authenticate the user to the device only. They do not authenticate the user to any remote service over the Internet, nor can they easily be extended to do so securely.

Other stronger, more technical authentication methods could be considered. Voters could be issued cryptographic ID cards such as the CAC cards issued to DoD personnel or like the national ID card of Estonia. Cryptographic ID cards would in principle enable voter authentication from any Internet-connected computer or device that could read them. But no U.S. state issues such IDs to its citizens or voters, and it seems unlikely that any will do so in the foreseeable future. Even if the security climate changes and people are willing to accept such an ID system, the startup and maintenance costs will be very high. Voters would have to buy computers or devices that could read the cards, and they would almost certainly have to be useful for other online purposes besides just voting in order to justify the costs involved to both the government and the voter.

The fact is that the U.S. has no strong, universally deployed online citizen identification and authentication system, and none is on the horizon. While strong remote voter authentication is not a fundamentally unsolvable problem, it is an immense practical problem that has to be dealt with before we can consider deploying any online voting system, including E2E-VIV systems.

Client side malware

In an E2E-VIV system voters compose and input their vote choices on a privately owned (hence unsecured) platform, either a PC or mobile device. If the voting platform is infected with malware or spyware, it is complicated to prevent the votes from being modified or reported to a third party, and impossible to prevent them from just being thrown away by the malware before the ballot is encrypted and enters the E2E-VIV system.

The malware threat is ubiquitous now and is fundamental to all online voting systems. No device is safe from malware infection. No software safeguards such as commercial antivirus systems are very effective against it. There are hundreds of ways that malware can infect a voting platform, sometimes by sophisticated technical means and sometimes by tricking users into taking unsafe actions. There are hundreds of places in the huge multilayered software ecosystem of a PC or mobile device where malware can hide and be launched from. And there are thousands of hardware, OS, browser, combinations, and countless configuration choices—way too many for any possible comprehensive defense against malware. A modern PC or mobile device can easily contain software elements from a hundred different companies or open source development groups, any one of which may either be malicious or contain critical vulnerabilities that enables malicious code. Such vulnerabilities are so numerous that more are discovered all the time and vendors release security updates on a regular basis to plug the more recently discovered holes. E2E-VIV systems have no ability to prevent or even detect the actions of malware before the votes are safely submitted into the E2E-VIV software.

Malware in voters' computers can undermine the election in three fundamental ways.

i) **Malware modification of votes:** Malware may actually modify the voter's choices surreptitiously, before they are submitted to the E2E-VIV system. The techniques for accomplishing this without tipping off the voter will depend on the detailed architecture of the voting system, e.g. whether the client side is packaged as a full-blown application, or as a mobile app, or a Javascript script, or a browser plugin, or some other form. But in all cases the voter's choices must be input to the PC or mobile device in the clear, unencrypted, and be processed by a large amount of system software and application/browser/script software *before* it is encrypted and submitted to the E2E-VIV system. Regardless of the E2E-VIV architecture, with today's software tools it is reasonably straightforward for malicious code to modify votes undetectably before they are encrypted.

   Depending on the design of the E2E-VIV system, it may be possible for some voters to discover after the fact that different votes were recorded for them than the ones they thought they cast. But even so, there will generally be no way to prove to election officials that the voter did not cast the votes recorded for her by mistake, or cast them deliberately and then change her mind. Whether there is a remedy for voters who claim their votes are modified by malware and falsely recorded is an unresolved question.

We cannot generally eradicate the threat of malware. But in the special case of online voting there are techniques that in theory can prevent malware from surreptitiously modifying votes. Unfortunately they all involve additional burdens on the voter in the form of code voting, or special hardware devices independent of the PC, or a second independent communication channel to the election server that does not use the Internet, or at least is guaranteed to use an independent path from the one the votes travel. All known methods of working around client side malware involve some complication in the voting process that will be enough of a barrier, at least for the time being, to discourage many voters.

ii) **Malware vote privacy violation:** Malware on a voter's computer or mobile device could allow her completed ballot to enter the E2EVIV system unmodified, but prior to that it could *also* send a copy of her votes to a third party. Unless a voter has considerable expertise and has special instrumentation running during the voting transaction there is no way for her to know it. And if the instrumentation was not in place before voting there is no after-the-fact test that can determine whether this happened, and certainly no way to reverse the privacy violation. If the voter is voting from a mobile device, as opposed to a PC, often no such instrumentation even exists today.

In any remote voting situation there is always the possibility that someone can physically look over a voter's shoulder and watch her vote. That is a risk we live with also with paper mail-in ballots. But the main concern is not with individual cases of privacy violation, but with widespread automated spying on many online votes.

Widespread vote privacy violation can undermine democracy in two major ways. First, in situations where some people have power over others (e.g. employers, commanding officers, union supervisors, parents, nursing home management) revealing who cast which ballot can be the basis for coercion or retaliation. This may not (yet) be a widespread concern in the U.S., but it certainly is in other countries.

Also, automated privacy violation can enable large scale, automated vote buying and selling. It is easy to imagine a scheme in which many voters are induced to sell their voting credentials, or to run a particular program while voting for a particular candidate in exchange for PayPal dollars or some other crypto currency such as Bitcoin that can be transmitted entirely online. The vote buying transaction would likely be totally undetectable by authorities. Even if the scheme eventually comes to the attention of authorities, the buyers may be long gone, or may be on foreign soil out of reach of U.S. law. In any case there would be no way to know how many votes were sold or who the sellers are. Technical tricks, such as allowing voters to vote multiple times online with only the last cast vote actually counting, are not effective when the attacker knows how the system works or when the voter cooperates in a vote sale.

As with malware that intends to modify ballots surreptitiously, there are workarounds that can prevent malware from surreptitiously revealing how someone votes to a third party. But again, they complicate the process of online voting sufficiently to be a barrier that will discourage many voters from voting

iii) **Malware denial of service:** The easiest and most intractable malware attack is one that simply prevents the voter from successfully voting. That can be done in many ways. The malware could make it appear that there was an error of some kind, which might be frustrating but hardly surprising to voters who might either blame themselves or their own flaky computers or attribute it to just another buggy online service. Alternatively, the malware might perfectly mimic a completed voting transaction, so that the voter believes she has successfully voted, whereas the malware would simply throw the ballot away.

Such a denial of service attack might not be very politically effective if it is applied to a random set of voters. But if it can be applied *selectively* to voters who would be likely to vote in a way that the attacker does not like, then it becomes a powerful partisan fraud tool. The malware writer may want to make a good guess as to how the voter will likely vote before deciding whether of not to block her from voting. Fortunately, there are many clues in a voter's computer or mobile device, such as browser history, to indicate at least a likely party preference or social class, and that is probably all the information the malware would need.

Some voters may be sophisticated enough to detect that their ballots were never included in the count, especially during the post-election verification stage when some might discover that there is no record that they voted. But any particular voter would find it almost impossible to prove to election officials that she actually tried to vote online but that malware prevented it. Perhaps she simply never really tried to vote, or

for some other technical reason not related to malware she had been prevented from successfully voting. There would be no evidence anywhere accessible to officials that could help them diagnose the situation. Even if the voter brought her computer in for forensic examination by experts, chances are that the malware would have erased evidence and erased itself, leaving no trace.

And finally, even if an obvious widespread malware denial of service attack were somehow discovered, there would be no way to estimate how many ballots were lost and how many voters were disenfranchised. The E2E-VIV system does nothing to help with such an estimate because the ballots are discarded by the malware before they ever enter the E2E-VIV system.

Unfortunately, while there are (at best inconvenient) workarounds for malware that aims to surreptitiously modify a ballot or send it to a third party, *there is fundamentally no workaround for malware designed to just prevent voting*. Well-designed malware would make the voter believe she had successfully voted, and she might never discover until it was too late that she did not. Even if she did discover it, the only recourse would be to vote from a different, uninfected PC or device, *but she almost certainly would not know that malware was the cause of the problem and would likely not know to vote from a different machine!*

Malware is a profound, absolutely fundamental problem that has been with us since the dawn of the PC age or before and will be with us for as far into the future as we can see. There is fundamentally no way to totally eradicate client side malware, or totally immunize against it, or even detect its presence. Malware is getting ever easier to write because templates, kits, libraries, and exemplars of successful malware are widely available to aid attackers, and because the payoff far exceeds the risks of getting caught. It is estimated that anywhere from 10 to 30 percent of all PCs in the world are infected with malware, and even more when spyware is included. And there are probably no reliable estimates as to the fraction of mobile devices similarly infected.

Finally, even if an easy to use, accessible workaround for client side malware is invented that preserves vote integrity and privacy, it will still not be possible to prevent a malware denial of service attack that just blocks voting. Such denial of service attacks are in a theoretically different category and there will never be a general way to thwart them all, nor a general way for voters or election officials to unambiguously recognize one. Even if it is recognized, there will be no way to estimate how many voters were affected.

The conclusion therefore, is that client side malware remains a fundamental threat that E2E-VIV systems cannot fully defend against.

Network attacks and Distributed Denial of Service (DDoS) attacks

E2E-VIV systems are client-server protocols that execute on top of several layers of other software, including the operating system and browser on the client side, the operating system and server complex on the server side, and the various levels of TCP/IP protocol stack all through the Internet, as well as routing protocols, DNS, NTP, DHCP, and also numerous other protocols used in wireless or mobile devices. The E2E-VIV system cannot work properly unless all of this other software works properly also. We have already discussed the problem of malicious logic on the client side. But E2E-VIV software is also attackable from the server side or from the Internet infrastructure software that the E2E-VIV software depends on.

There are many ways to attack an E2E-VIV election by maliciously modifying or configuring the software in the Internet. Such attacks are called *network attacks*. Any IT person who controls a router, DNS server, or another element of Internet infrastructure is in a position to prevent votes from getting to their destinations. On the positive side, E2E-VIV systems are partly robust against such network attacks in that they cannot result in votes being falsely injected or modified without detection. This is a clear advantage that E2E-VIV systems have over other Internet voting systems. However, there is no way to prevent a network attack from disrupting the E2E-VIV protocols in a way that causes ballots to be lost, i.e. undelivered. While this will also be detectable, the malicious loss of votes in transit cannot be prevented by an E2E-VIV protocol, and it may not be possible even to estimate the number of votes affected.

One especially dangerous form of network attack is the *distributed denial of service* (DDoS) attack. In this attack, an attacker floods the server (or some other subsystem) with so much traffic or other work that it either crashes the system or else slows it to such a crawl that it is effectively down. Voters would experience a DDoS attack on a vote server as either *extreeeeemly* long waits between steps in the voting process, or total nonresponsiveness of the server, or some other error. The net effect is that large numbers of voters would simply be disenfranchised. The attack can be directed pointedly at the server side, in which case all online voters would be affected, or it could be selectively directed at certain parts of the Internet infrastructure that would affect only a subset of the voters.

We have to be able to defend online elections against DDoS attacks for two key reasons. First, they are about the easiest of all network attacks to perpetrate. There are many different kinds of DDoS attack against different parts of Internet infrastructure and different levels of software, and there are many kits available on the dark net to allow anyone from anywhere in the world to perpetrate a DDoS attack against almost any target. In fact, the means of DDoS attack are so routinized and ubiquitous that there are illegal businesses online that will conduct an attack to your specifications against any target you choose for a moderate price. You can ask for, say, a 50 gigabit per second attack for the last 4 hours on Election Day against the IP address of the (hypothetical) Cook County vote server. That would probably prevent anyone from voting online in that jurisdiction during those hours. All of those voters would be disenfranchised, but none of them would be able to prove that they were among of the victims, and election officials would not even be able to make a decent estimate of the number of ballots lost.

DDoS attacks have actually been used in real public elections around the world at least four separate times that have been made public. (Arizona Democratic Primary, 2000; Ontario NDP, 2003; Hong Kong people's election, 2012; NDP of Canada 2012). While there are various tools that can be used to *ameliorate* some DDoS attacks, there is no *general solution*, and the DDoS problem is so fundamental that there will probably never be one with the current architecture of the Internet. Vulnerability to DDoS attacks is effectively built in to its design. Hence, all E2E-VIV systems are vulnerable to network attacks that can result in disenfranchising a large number voters with no way of even measuring how many were affected. There is no fundamental defense against DDoS attacks.

A.2.5 Conclusion

E2E-V offers a dramatic improvement in the security of voting systems. While it is necessary for *any* online voting system for public elections, but it is by no means sufficient. Once it is embedded in a larger *Internet voting* context fundamental new security vulnerabilities appear for which there are no solutions today, and no prospect of solutions in the foreseeable future. These include vulnerability to authentication attacks, client side malware attacks, and DDoS attacks that can be perpetrated by anyone in the world. Unless and until those additional security problems are satisfactorily and simultaneously solved—and they may never be—we must not consider any Internet voting system for use in public elections.

Appendix B

Usability Study Report

Editorial note: During layout we will drop in the text of the report and cite the original PDF version as well, which will be provided as a separately downloadable artifact on the project website.

Bibliography

[1] T. Abdoul et al. “AADL Execution Semantics Transformation for Formal Verification”. In: *13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2008)*. Mar. 2008, pp. 263–268. DOI: 10.1109/ICECCS.2008.24.

[2] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[3] *Accessible Voting Technology Initiative*. URL: http://elections.itif.org/ (visited on 07/01/2015).

[4] *Accessible Voting Technology Initiative: Election Design Guidelines*. URL: http://elections.itif.org/resources/guidelines/ (visited on 07/01/2015).

[5] Claudia Z. Acemyan et al. “Usability of Voter Verifiable, End-to-end Voting Systems: Baseline Data for Helios, Prêt à Voter, and Scantegrity II”. In: *The USENIX Journal of Election Technology and Systems* (2014), p. 26.

[6] *ACSL: ANSI ISO C Specification Language*. URL: http://www.frama-c.com/download/acsl.pdf (visited on 07/01/2015).

[7] *ADA Standards for Accessible Design*. URL: http://www.ada.gov/2010ADAstandards_index.htm (visited on 07/01/2015).

[8] Ben Adida. “Helios: Web-based Open-Audit Voting”. In: *USENIX Security*. 2008. URL: https://www.usenix.org/legacy/events/sec08/tech/full_papers/adida/adida.pdf (visited on 07/01/2015).

[9] Ben Adida et al. “Electing a University President using Open-Audit Voting: Analysis of real-world use of Helios”. In: *USENIX EVT/WOTE*. 2009. URL: https://www.usenix.org/legacy/event/evtwote09/tech/full_papers/adida-helios.pdf (visited on 07/01/2015).

[10] *Alloy: A Language & Tool for Relational Models*. URL: http://alloy.mit.edu/ (visited on 07/01/2015).

[11] *American Fuzzy Lop*. URL: http://lcamtuf.coredump.cx/afl/ (visited on 07/01/2015).

[12] *Ansible*. URL: http://www.ansible.com/ (visited on 07/01/2015).

[13] Tigran Antonyan et al. “State-wide elections, optical scan voting systems, and the pursuit of integrity”. In: *Information Forensics and Security, IEEE Transactions on* 4.4 (2009), pp. 597–610.

[14] *Anywhere Ballot*. URL: http://anywhereballot.com/ (visited on 07/01/2015).

[15] *Apache Bloodhound*. URL: http://bloodhound.apache.org/ (visited on 07/01/2015).

[16] *Apache Maven Project*. URL: http://maven.apache.org/ (visited on 07/01/2015).

[17] *ARINC Standards 600 Series*. URL: http://store.aviation-ia.com/cf/store/catalog.cfm?prod_group_id=1&category_group_id=3 (visited on 07/01/2015).

[18] Alessandro Armando et al. “The AVISPA tool for the automated validation of internet security protocols and applications”. In: *Computer Aided Verification*. Springer. 2005, pp. 281–285.

[19] *asm.js: an extraordinarily optimizable, low-level subset of JavaScript*. URL: http://www.asmjs.org/ (visited on 07/01/2015).

[20] American Political Science Association et al. “Findings and recommendations of the special committee on service voting”. In: *American Political Science Review* 46.2 (1952), pp. 512–523.

[21] Atlassian. *Comparing Workflows*. URL: https://www.atlassian.com/git/tutorials/comparing-workflows (visited on 07/01/2015).

[22] Dave Bayer, Stuart Haber, and W. Scott Stornetta. "Improving the Efficiency and Reliability of Digital Time-Stamping". In: *Sequences II: Methods in Communication, Security and Computer Science*. Springer-Verlag, 1993, pp. 329–334.

[23] Jonathan Ben-Nun et al. "A new implementation of a dual (paper and cryptographic) voting system". In: *Electronic Voting*. 2012, pp. 315–329.

[24] Josh Benaloh and Michael de Mare. *Efficient Broadcast Time-Stamping*. Tech. rep. Jan. 1991. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=68476.

[25] Cedric Beust and Hani Suleiman. *Next Generation Java Testing*. Addison-Wesley, 2007.

[26] David Bismark et al. "Experiences Gained from the first Prêt à Voter Implementation". In: *First International Workshop on Requirements Engineering for e-Voting Systems (RE-VOTE)*. IEEE. Aug. 2009, pp. 19–28. DOI: 10.1109/RE-VOTE.2009.5.

[27] *BitBucket*. URL: http://www.bitbucket.org/ (visited on 07/01/2015).

[28] Bruno Blanchet. "Automatic Verification of Security Protocols in the Symbolic Model: The Verifier ProVerif". English. In: *Foundations of Security Analysis and Design VII*. Ed. by Alessandro Aldini, Javier Lopez, and Fabio Martinelli. Vol. 8604. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 54–87. ISBN: 978-3-319-10081-4. DOI: 10.1007/978-3-319-10082-1_3.

[29] Günter Böckle et al. "Calculating ROI for software product lines". In: *Software* 21.3 (2004), pp. 23–31.

[30] Dan Boneh, Amit Sahai, and Brent Waters. "Functional encryption: Definitions and challenges". In: *Theory of Cryptography*. Springer, 2011, pp. 253–273.

[31] Jan Bosch. "Maturity and evolution in software product lines: Approaches, artefacts and organization". In: *Software Product Lines*. Springer, 2002, pp. 257–271.

[32] Lionel C. Briand et al. "Exploring the relationships between design measures and software quality in object-oriented systems". In: *Journal of systems and software* 51.3 (2000), pp. 245–273.

[33] Philippe Bulens, Damien Giry, and Olivier Pereira. "Running mixnet-based elections with Helios". In: *USENIX EVT/WOTE*. 2011. URL: http://www.usenix.org/events/evtwote11/tech/final_files/Bulens.pdf (visited on 07/01/2015).

[34] Craig Burton et al. "Using Prêt à Voter in Victorian State elections". In: *USENIX EVT/WOTE*. 2012. URL: https://www.usenix.org/system/files/conference/evtwote12/evtwote12-final9_0.pdf (visited on 07/01/2015).

[35] Michael D. Byrne, Kristen K. Greene, and Sarah P. Everett. "Usability of voting systems: Baseline data for paper, punch cards, and lever machines". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 2007, pp. 171–180.

[36] *C0: Specification and Verification in Introductory Computer Science*. URL: http://c0.typesafety.net/ (visited on 07/01/2015).

[37] Richard Carback et al. "Scantegrity II Municipal Election at Takoma Park: The First E2E Binding Governmental Election with Ballot Privacy". In: *USENIX Security*. 2010. URL: https://www.usenix.org/legacy/events/sec10/tech/full_papers/Carback.pdf (visited on 07/01/2015).

[38] Carter Center. *Internet Voting Pilot: Norway's 2013 Parliamentary Elections*. Mar. 2014. URL: http://www.cartercenter.org/resources/pdfs/peace/democracy/Carter-Center-Norway-2013-study-mission-report2.pdf (visited on 07/01/2015).

[39] *Center for Advanced Software Analysis: Software Tools*. URL: http://casa.au.dk/software-tools/ (visited on 07/01/2015).

[40] Scott Chacon. *Pro Git*. Apress, 2014. ISBN: 978-1484200773.

[41] David Chaum, Claude Crépeau, and Ivan Damgard. "Multiparty unconditionally secure protocols". In: *Proceedings of the 20th Annual ACM Symposium on Theory of Computing (STOC)*. ACM. 1988, pp. 11–19.

[42] David Chaum, Peter Y. A. Ryan, and Steve Schneider. "A Practical Voter-Verifiable Election Scheme". English. In: *Computer Security – ESORICS 2005*. Ed. by Sabrinade Capitani di Vimercati, Paul Syverson, and Dieter Gollmann. Vol. 3679. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 118–139. ISBN: 978-3-540-28963-0. DOI: 10.1007/11555827_8.

[43] David Chaum et al. "Accessible voter-verifiability". In: *Cryptologia* 33.3 (2009), pp. 283–291.

[44] David Chaum et al. "Scantegrity II: End-to-End Verifiability by Voters of Optical Scan Elections Through Confirmation Codes". In: *IEEE Transactions on Information Forensics and Security* 4.4 (Dec. 2009), pp. 611–627. ISSN: 1556-6013. DOI: 10.1109/TIFS.2009.2034919.

[45] David Chaum et al. "Scantegrity II: End-to-End Verifiability for Optical Scan Election Systems using Invisible Ink Confirmation Codes". In: *USENIX EVT*. 2008. URL: https://www.usenix.org/legacy/event/evt08/tech/full_papers/chaum/chaum.pdf (visited on 07/01/2015).

[46] David Chaum et al. "Scantegrity II: end-to-end verifiability for optical scan election systems using invisible ink confirmation codes". In: *EVT* 8 (2008), pp. 1–13.

[47] *Checkstyle*. URL: http://checkstyle.sourceforge.net/ (visited on 07/01/2015).

[48] Lianping Chen, Muhammad Ali Babar, and Nour Ali. "Variability management in software product lines: a systematic review". In: *Proceedings of the 13th International Software Product Line Conference*. Carnegie Mellon University. 2009, pp. 81–90.

[49] *Clafer: Lightweight Modeling Language*. URL: http://www.clafer.org/ (visited on 07/01/2015).

[50] Jeremy Clark and Aleksander Essex. "CommitCoin: carbon dating commitments with Bitcoin". In: *Financial Cryptography and Data Security*. Springer, 2012, pp. 390–398.

[51] Paul Clements and Linda Northrop. *Software product lines: practices and patterns*. Addison-Wesley, 2002.

[52] *CloudFlare, Inc.* URL: http://www.cloudflare.com/ (visited on 07/01/2015).

[53] Alistair Cockburn and Laurie Williams. "The costs and benefits of pair programming". In: *Extreme programming examined* (2000), pp. 223–247.

[54] *Code Contracts*. URL: http://research.microsoft.com/en-us/projects/contracts/ (visited on 07/01/2015).

[55] U.S. Election Assistance Commission. *Election Administration and Voting Survey*. URL: http://www.eac.gov/research/election_administration_and_voting_survey.aspx (visited on 07/01/2015).

[56] *Community Z Tools: Tools for Developing and Reasoning About Z Specifications*. URL: http://czt.sourceforge.net/ (visited on 07/01/2015).

[57] *CompCert*. URL: http://compcert.inria.fr/ (visited on 07/01/2015).

[58] Ronald Cramer, Ivan Damgård, and Ueli Maurer. "General Secure Multi-party Computation from any Linear Secret-Sharing Scheme". English. In: *Advances in Cryptology—EUROCRYPT 2000*. Ed. by Bart Preneel. Vol. 1807. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2000, pp. 316–334. ISBN: 978-3-540-67517-4. DOI: 10.1007/3-540-45539-6_22.

[59] Cas J. F. Cremers. "The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols". In: *Computer Aided Verification*. Springer, 2008, pp. 414–418.

[60] *CryptoVerif: Cryptographic Protocol Verifier in the Computational Model*. URL: http://cryptoverif.inria.fr/ (visited on 07/01/2015).

[61] Chris Culnane et al. "vVote: a Verifiable Voting System". In: *arXiv preprint arXiv:1404.6822* (2014). URL: http://arxiv.org/abs/1404.6822 (visited on 07/01/2015).

[62] *CVK: Crypto Verification Kit*. URL: http://research.microsoft.com/en-us/projects/cvk/ (visited on 07/01/2015).

[63] Krzysztof Czarnecki et al. "Model-driven software product lines". In: *Companion to the 20th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM. 2005, pp. 126–127.

[64] *Dafny: a language and program verifier for functional correctness*. URL: http://research.microsoft.com/en-us/projects/dafny/ (visited on 07/01/2015).

[65] Alex Delis et al. *Pressing the Button for European Elections 2014: Public attitudes towards Verifiable E-Voting In Greece*. June 2014. URL: https://drive.google.com/file/d/0B-mtbRwyPn_SdnpMRzBKcEZWUm8/view?usp=sharing (visited on 07/01/2015).

[66] Jared DeMott. "Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing". In: *BlackHat and DefCon*. 2007.

[67] David L. Detlefs et al. *SRC Research Report 159: Extended Static Checking*. Compaq Systems Research Center, Dec. 1998.

[68] *DO-178B Software Considerations in Airborne Systems and Equipment Certification*. URL: http://www.rtca.org/store_product.asp?prodid=581 (visited on 07/01/2015).

[69] *DO-178C Software Considerations in Airborne Systems and Equipment Certification*. URL: http://www.rtca.org/store_product.asp?prodid=803 (visited on 07/01/2015).

[70] *Docker*. URL: http://www.docker.com/ (visited on 07/01/2015).

[71] *EasyCrypt: Computer-Aided Cryptographic Proofs*. URL: http://www.easycrypt.info/ (visited on 07/01/2015).

[72] *Eiffel Inspector*. URL: https://docs.eiffel.com/book/eiffelstudio/eiffel-inspector (visited on 07/01/2015).

[73] *EMMA: A free Java code coverage tool*. URL: http://emma.sourceforge.net/ (visited on 07/01/2015).

[74] Michael D. Ernst et al. "The Daikon system for dynamic detection of likely invariants". In: *Science of Computer Programming* 69.1–3 (2007), pp. 35–45. DOI: 10.1016/j.scico.2007.01.015.

[75] Aleks Essex et al. "Punchscan in practice: an E2E election case study". In: *Proceedings of Workshop on Trustworthy Elections*. 2007.

[76] *F*: A higher-order effectual language designed for program verification*. URL: https://fstar-lang.org (visited on 07/01/2015).

[77] *FACE Technical Standard Edition 2.1*. URL: http://www.opengroup.org/face/tech-standard-2.1 (visited on 07/01/2015).

[78] Michael Fagan. "Design and code inspections to reduce errors in program development". In: *Software pioneers*. Springer, 2002, pp. 575–607.

[79] *Federal Information Processing Standards Publication 140-2 - Security Requirements for Cryptographic Modules*. URL: http://csrc.nist.gov/publications/fips/fips140-2/fips1402.pdf (visited on 07/01/2015).

[80] Peter H. Feiler and David P. Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis and Design Language*. Addison-Wesley Professional, Sept. 2013.

[81] *FindBugs*. URL: http://findbugs.sourceforge.net/ (visited on 07/01/2015).

[82] *FogBugz*. URL: http://www.fogcreek.com/fogbugz/ (visited on 07/01/2015).

[83] Python Software Foundation. *doctest — Test interactive Python examples*. 2015. URL: https://docs.python.org/3/library/doctest.html (visited on 07/01/2015).

[84] *Frama-C*. URL: http://www.frama-c.com/ (visited on 07/01/2015).

[85] Jerry Zeyu Gao, H.-S. Jacob Tsao, and Ye Wu. *Testing and quality assurance for component-based software*. Artech House, 2002.

[86] Federal Constitutional Court of Germany. *Docket Nos. 2 BvC 3/07 & 2 BvC 4/07*. 2009. URL: https://www.bundesverfassungsgericht.de/SharedDocs/Entscheidungen/EN/2009/03/cs20090303_2bvc000307en.html (visited on 07/01/2015).

[87] *GitHub*. URL: http://www.github.org/ (visited on 07/01/2015).

[88] Kristian Gjøsteen. "The Norwegian Internet Voting Protocol". English. In: *E-Voting and Identity*. Ed. by Aggelos Kiayias and Helger Lipmaa. Vol. 7187. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 1–18. ISBN: 978-3-642-32746-9. DOI: 10.1007/978-3-642-32747-6_1.

[89] Milos Gligoric et al. “Comparing non-adequate test suites using coverage criteria”. In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM. 2013, pp. 302–313.

[90] *GNATcoverage Coverage Analysis Tool*. URL: http://www.adacore.com/gnatcovarage/ (visited on 07/01/2015).

[91] *GNU Make*. URL: http://www.gnu.org/software/make/ (visited on 07/01/2015).

[92] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. “Grammar-based Whitebox Fuzzing”. In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’08. Tucson, AZ, USA: ACM, 2008, pp. 206–215. ISBN: 978-1-59593-860-2. DOI: 10.1145/1375581.1375607.

[93] Shafi Goldwasser, Silvio Micali, and Avi Wigderson. “How to play any mental game, or a completeness theorem for protocols with an honest majority”. In: *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC)*. 1987, pp. 218–229.

[94] Rop Gonggrijp et al. “RIES - Rijnland Internet Election System: A Cursory Study of Published Source Code”. English. In: *E-Voting and Identity*. Ed. by Peter Y.A. Ryan and Berry Schoenmakers. Vol. 5767. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 157–171. ISBN: 978-3-642-04134-1. DOI: 10.1007/978-3-642-04135-8_10.

[95] Vipul Goyal et al. “Attribute-based encryption for fine-grained access control of encrypted data”. In: *Proceedings of the 13th ACM Conference on Computer and Communications Security*. Acm. 2006, pp. 89–98.

[96] *Gradle*. URL: http://www.gradle.org/ (visited on 07/01/2015).

[97] Trusted Computing Group. *TPM Main Specification*. URL: http://www.trustedcomputinggroup.org/resources/tpm_main_specification (visited on 07/01/2015).

[98] *Guide for Applying the Risk Management Framework to Federal Information Systems*. URL: http://csrc.nist.gov/publications/nistpubs/800-37-rev1/sp800-37-rev1-final.pdf (visited on 07/01/2015).

[99] Stuart Haber and W. Scott Stornetta. “How to Time-Stamp a Digital Document”. In: *Advances in Cryptology—CRYPT0’ 90*. Ed. by Alfred J. Menezes and Scott A. Vanstone. Vol. 537. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1991, pp. 437–455. ISBN: 978-3-540-54508-8. DOI: 10.1007/3-540-38424-3_32.

[100] Øystein Haugen, Andrzej Wąsowski, and Krzysztof Czarnecki. “CVL: Common Variability Language”. In: *Proceedings of the 16th International Software Product Line Conference-Volume 2*. ACM. 2012, pp. 266–267.

[101] James Herbsleb et al. “Software quality and the capability maturity model”. In: *Communications of the ACM* 40.6 (1997), pp. 30–40.

[102] *HOL4*. URL: http://hol.sourceforge.net.

[103] Gerard J. Holzmann. *UNO: Static source code checking for user-defined properties*. 2002. URL: http://www.spinroot.com/uno/uno_long.pdf (visited on 07/01/2015).

[104] *How to Vote: Wombat Voting System*. URL: http://www.wombat-voting.com/how-to-vote (visited on 07/01/2015).

[105] Engelbert Hubbers, Bart Jacobs, and Wolter Pieters. “RIES - Internet Voting in Action”. In: *29th International Computer Software and Applications Conference (COMPSAC 2005)*. IEEE. 2005, pp. 417–424.

[106] Engelbert Hubbers et al. “Description and analysis of the RIES internet voting system”. In: *Report of the Eidhoven Institute for the Protection of Systems and Information*. Faculty of Mathematics and Computer Science Eindhoven University of Technology, June 2008.

[107] *HUnit-Plus: A test framework building on HUnit*. URL: https://hackage.haskell.org/package/HUnit-Plus (visited on 07/01/2015).

[108] *IEC 62304 - Medical Device Software - Software life cycle processes*. URL: http://www.iso.org/iso/catalogue_detail.htm?csnumber=38421 (visited on 07/01/2015).

[109] *IEEE 1622-2011 - IEEE Standard for Electronic Distribution of Blank Ballots for Voting Systems*. URL: http://ieeexplore.ieee.org/servlet/opac?punumber=6130554 (visited on 07/01/2015).

[110] Laura Inozemtseva and Reid Holmes. “Coverage is not strongly correlated with test suite effectiveness”. In: *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. ACM. 2014, pp. 435–445.

[111] *International Technology and Persons with Disabilities Conference*. URL: http://www.csun.edu/cod/conference/index.php (visited on 07/01/2015).

[112] *Isabelle*. URL: https://isabelle.in.tum.de/.

[113] *ISO 9000 - Quality Management*. URL: http://www.iso.org/iso/home/standards/management-standards/iso_9000.htm (visited on 07/01/2015).

[114] *ISO/IEC 15408-1*. URL: http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=50341 (visited on 07/01/2015).

[115] *ISO/IEC 15408-1*. URL: http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=46414 (visited on 07/01/2015).

[116] *ISO/IEC 15408-1*. URL: http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=46413 (visited on 07/01/2015).

[117] *Jenkins: An Extensible Open Source Continuous Integration Server*. URL: http://jenkins-ci.org/ (visited on 07/01/2015).

[118] *JIRA*. URL: https://www.atlassian.com/software/jira (visited on 07/01/2015).

[119] *JSCert: Certified JavaScript*. URL: http://www.jscert.org/ (visited on 07/01/2015).

[120] *JUnit*. URL: http://www.junit.org/ (visited on 07/01/2015).

[121] Stephen H. Kan. *Metrics and models in software quality engineering*. Addison-Wesley, 2002.

[122] K.C. Kang et al. *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. CMU/SEI-90-TR-021. Software Engineering Institute, Carnegie Mellon University, Nov. 1990.

[123] Christian Kästner, Sven Apel, and Martin Kuhlemann. "Granularity in software product lines". In: *Proceedings of the 30th International Conference on Software Engineering (ICSE)*. ACM. 2008, pp. 311–320.

[124] James E. King and Rhod J. Jones. *Trusted platform modules*. US Patent 8,429,423. Apr. 2013.

[125] Joseph R. Kiniry and Daniel M. Zimmerman. *E2E-VIV Domain Model*. URL: http://www.usvotefoundation.org/E2E-VIV/domain-model (visited on 07/10/2015).

[126] Joseph R. Kiniry and Daniel M. Zimmerman. *E2E-VIV Requirements*. URL: http://www.usvotefoundation.org/E2E-VIV/requirements (visited on 07/10/2015).

[127] Gerwin Klein et al. "seL4: Formal verification of an OS kernel". In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM. 2009, pp. 207–220.

[128] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". In: *ACM Trans. Program. Lang. Syst.* 4.3 (July 1982), pp. 382–401. ISSN: 0164-0925. DOI: 10.1145/357172.357176.

[129] John Launchbury et al. "Application-Scale Secure Multiparty Computation". In: *Programming Languages and Systems*. Ed. by Zhong Shao. Vol. 8410. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 8–26. ISBN: 978-3-642-54832-1. DOI: 10.1007/978-3-642-54833-8_2.

[130] Gary T. Leavens et al. *JML Reference Manual*. 2013. URL: http://www.jmlspecs.org/refman/jmlrefman.pdf (visited on 07/01/2015).

[131] Seunghyun Lee et al. "EZ ballot with multimodal inputs and outputs". In: *Proceedings of the 14th International ACM SIGACCESS Conference on Computers and Accessibility*. ACM. 2012, pp. 215–216.

[132] J.R. Lewis and B. Martin. "Cryptol: high assurance, retargetable crypto development and validation". In: *Military Communications Conference, 2003. MILCOM '03. 2003 IEEE*. Vol. 2. Oct. 2003, pp. 820–825. DOI: 10.1109/MILCOM.2003.1290218.

[133] Frank J Linden, Klaus Schmid, and Eelco Rommes. *Software product lines in action: the best industrial practice in product line engineering*. Springer Science & Business Media, 2007.

[134] M.V. Linhares et al. "Introducing the modeling and verification process in SysML". In: *IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2007)*. Sept. 2007, pp. 344–351. DOI: 10.1109/EFTA.2007.4416788.

[135] Gavin Lowe. "Casper: A compiler for the analysis of security protocols". In: *Computer Security Foundations Workshop*. IEEE. 1997, pp. 18–30.

[136] Anil Madhavapeddy et al. “Unikernels: Library Operating Systems for the Cloud”. In: *SIGPLAN Notices* 48.4 (Mar. 2013), pp. 461–472. ISSN: 0362-1340. DOI: 10.1145/2499368.2451167.

[137] Neal McBurnett et al. *Scantegrity Responds to Rice Study on Usability of the Scantegrity II Voting System*. Dec. 2014. URL: http://vote.caltech.edu/content/scantegrity-responds-rice-study-usability-scantegrity-ii-voting-system (visited on 07/01/2015).

[138] David Kaloper Mersinjak et al. “Not-quite-so-broken TLS: Lessons in Re-engineering a Security Protocol Specification and Implementation”. In: *High Confidence Software and Systems Conference*. 2015.

[139] *Méthode B*. URL: http://www.methode-b.com/ (visited on 07/01/2015).

[140] Bertrand Meyer. *Object-Oriented Software Construction*. 2nd Edition. Prentice-Hall, 1997.

[141] Bertrand Meyer. “The grand challenge of trusted components”. In: *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*. 2003, pp. 660–667.

[142] Bertrand Meyer et al. “Programs That Test Themselves”. In: *Computer* 42.9 (Sept. 2009), pp. 46–55. ISSN: 0018-9162. DOI: 10.1109/MC.2009.296.

[143] Michael P. McDonald. *United States Elections Project*. URL: http://www.electproject.org/.

[144] *Mirage OS*. URL: http://www.openmirage.org/ (visited on 07/01/2015).

[145] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. “The influence of organizational structure on software quality: an empirical case study”. In: *Proceedings of the 30th International Conference on Software Engineering (ICSE)*. ACM. 2008, pp. 521–530.

[146] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. URL: http://bitcoin.org/bitcoin.pdf.

[147] George C. Necula. “Proof-Carrying Code. Design and Implementation”. In: *Proof and System-Reliability*. Ed. by Helmut Schwichtenberg and Ralf Steinbrüggen. Vol. 62. NATO Science Series. Springer Netherlands, 2002, pp. 261–288. ISBN: 978-1-4020-0608-1. DOI: 10.1007/978-94-010-0413-8_8.

[148] *NUnit*. URL: http://www.nunit.org/ (visited on 07/01/2015).

[149] Object Management Group. *UML Human-Usable Textual Notation*. Aug. 2004. URL: http://www.omg.org/spec/HUTN/1.0/ (visited on 07/01/2015).

[150] United States General Accounting Office. *Voters with disabilities: access to polling places and alternative voting methods*. 2001. URL: http://www.gao.gov/products/GAO-02-107 (visited on 07/01/2015).

[151] *OMG Systems Modeling Language*. URL: http://www.omgsysml.org/ (visited on 07/01/2015).

[152] *OpenCover*. URL: https://github.com/OpenCover/opencover (visited on 07/01/2015).

[153] *OpenJML*. URL: http://openjml.org/ (visited on 07/01/2015).

[154] *OSATE2-Ocarina*. URL: http://www.openaadl.org/osate-ocarina.html (visited on 07/01/2015).

[155] *OSCE/ODIHR Election Assessment Mission Report*. Nov. 2006. URL: http://www.osce.org/odihr/elections/netherlands/24322?download=true (visited on 07/01/2015).

[156] *Overture Tool: Formal Modeling in VDM*. URL: http://overturetool.org/ (visited on 07/01/2015).

[157] *PCI SSC Data Security Standards Overview*. URL: https://www.pcisecuritystandards.org/security_standards/ (visited on 07/01/2015).

[158] Ronald Perez, Reiner Sailer, Leendert van Doorn, et al. “vTPM: virtualizing the trusted platform module”. In: *Proc. 15th Conf. on USENIX Security Symposium*. 2006, pp. 305–320.

[159] C Pilato. *Version control with Subversion*. Sebastopol, CA: O’Reilly Media, 2008. ISBN: 0596510330.

[160] *PMD*. URL: http://pmd.sourceforge.net/ (visited on 07/01/2015).

[161] Raluca Ada Popa et al. “CryptDB: Protecting Confidentiality with Encrypted Query Processing”. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP ’11. Cascais, Portugal: ACM, 2011, pp. 85–100. ISBN: 978-1-4503-0977-6. DOI: 10.1145/2043556.2043566.

[162] Stefan Popoveniuc and Ben Hosp. “An introduction to Punchscan”. In: *IAVoSS Workshop On Trustworthy Elections (WOTE 2006)*. Robinson College United Kingdom. 2006, pp. 28–30.

[163] Stefan Popoveniuc and Ben Hosp. “An Introduction to PunchScan”. English. In: *Towards Trustworthy Elections*. Ed. by David Chaum et al. Vol. 6000. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 242–259. ISBN: 978-3-642-12979-7. DOI: 10.1007/978-3-642-12980-3_15.

[164] *Puppet Labs*. URL: http://www.puppetlabs.com/ (visited on 07/01/2015).

[165] *QuickCheck: Automatic testing of Haskell programs*. URL: https://hackage.haskell.org/package/QuickCheck (visited on 07/01/2015).

[166] *RAISE—Rigorous Approach to Industrial Software Engineering*. URL: http://spd-web.terma.com/Projects/RAISE/ (visited on 07/01/2015).

[167] *Redmine*. URL: http://www.redmine.org/ (visited on 07/01/2015).

[168] *ReSharper*. URL: https://www.jetbrains.com/resharper/ (visited on 07/01/2015).

[169] Ronald L. Rivest and John P. Wack. *On the notion of “software independence” in voting systems*. Prepared for the TGDC, and posted by NIST at the given URL. July 2006. URL: http://vote.nist.gov/SI-in-voting.pdf.

[170] Noel H. Runyan. *Improving access to voting: A report on the technology for accessible voting systems*. 2007. URL: http://www.demos.org/publication/improving-access-voting-report-technology-accessible-voting-systems (visited on 07/01/2015).

[171] Peter Y.A. Ryan et al. “Prêt à Voter: a Voter-Verifiable Voting System”. In: *Information Forensics and Security, IEEE Transactions on* 4.4 (Dec. 2009), pp. 662–673. ISSN: 1556-6013. DOI: 10.1109/TIFS.2009.2033233.

[172] *SAW: The Software Analysis Workbench*. URL: http://saw.galois.com/ (visited on 07/01/2015).

[173] *Section508.gov: Opening Doors to IT*. URL: https://www.section508.gov (visited on 07/01/2015).

[174] *Security and Privacy Controls for Federal Information Systems and Organizations*. URL: http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r4.pdf (visited on 07/01/2015).

[175] *Security Review: Helios Online Voting*. Mar. 2009. URL: https://cubist.cs.washington.edu/Security/2009/03/13/security-review-helios-online-voting/ (visited on 07/01/2015).

[176] Adi Shamir. “Identity-based cryptosystems and signature schemes”. In: *Advances in cryptology*. Springer. 1985, pp. 47–53.

[177] N. Shankar. “PVS: Combining specification, proof checking, and model checking”. In: *Formal Methods in Computer-Aided Design*. Ed. by Mandayam Srivas and Albert Camilleri. Vol. 1166. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1996, pp. 257–264. ISBN: 978-3-540-61937-6. DOI: 10.1007/BFb0031813.

[178] Claire M. Smith. *Convenience Voting and Technology: The Case of Military and Overseas Voters (Elections, Voting, Technology)*. Palgrave Macmillan, 2014. ISBN: 1137398582.

[179] Claire M. Smith. *Time to Move: Overseas and Military Voter State Policy Innovation*. 2011. URL: https://www.overseasvotefoundation.org/files/Time_to_MOVE_March2011.doc (visited on 07/01/2015).

[180] *SourceForge*. URL: http://www.sourceforge.net/ (visited on 07/01/2015).

[181] *SPARK 2014*. URL: http://www.spark-2014.org/ (visited on 07/01/2015).

[182] *SPARK Pro*. URL: http://www.adacore.com/sparkpro/ (visited on 07/01/2015).

[183] Evan R. Sparks. *A Security Assessment of Trusted Platform Modules*. Tech. rep. TR2007-597. Department of Computer Science, Dartmouth College, 2007. URL: http://www.cs.dartmouth.edu/reports/abstracts/TR2007-597/ (visited on 07/01/2015).

[184] *StyleCop*. URL: http://www.stylecop.com/ (visited on 07/01/2015).

[185] *SUnit: The mother of all unit testing frameworks*. URL: http://sunit.sourceforge.net/ (visited on 07/01/2015).

[186] Wouter Swierstra. “Xmonad in Coq (experience report): Programming a window manager in a proof assistant”. In: *ACM SIGPLAN Notices*. Vol. 47. 12. ACM. 2012, pp. 131–136.

[187] *The Coq Proof Assistant*. URL: https://coq.inria.fr/ (visited on 07/01/2015).

[188] *The Haskell Cabal: Common Architecture for Building Applications and Libraries*. URL: https://www.haskell.org/cabal/ (visited on 07/01/2015).

[189] *The Haskell Lightweight Virtual Machine (HaLVM)*. URL: https://galois.com/project/halvm/ (visited on 07/01/2015).

[190] Ken Thompson. "Reflections on Trusting Trust". In: *Communications of the ACM* 27.8 (Aug. 1984), pp. 761–763. ISSN: 0001-0782. DOI: 10.1145/358198.358210.

[191] Nikolai Tillmann and Jonathan de Halleux. "Pex—White Box Test Generation for .NET". In: *Proc. of Tests and Proofs (TAP'08)*. Vol. 4966. Lecture Notes in Computer Science. Prato, Italy: Springer-Verlag, Apr. 2008, pp. 134–153. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=81193 (visited on 07/01/2015).

[192] James E Tomayko. "A comparison of pair programming to inspections for software defect reduction". In: *Computer Science Education* 12.3 (2002), pp. 213–222.

[193] *Trac*. URL: http://trac.edgewall.org/ (visited on 07/01/2015).

[194] Georgios Tsoukalas et al. "From Helios to Zeus". In: *USENIX Journal of Election Technology and Systems* 1.1 (Aug. 2013). URL: https://www.usenix.org/jets/issues/0101/tsoukalas (visited on 07/01/2015).

[195] *United States Census Bureau*. URL: http://www.census.gov/ (visited on 07/01/2015).

[196] *United States Election Assistance Commission: Accredited Test Laboratories*. URL: http://www.eac.gov/testing_and_certification/accredited_test_laboratories.aspx (visited on 07/01/2015).

[197] *UPPAAL*. URL: http://www.uppaal.org/ (visited on 07/01/2015).

[198] U.S. Election Assistance Commission. *UOCAVA Pilot Program Testing Requirements—August 25, 2010*. Aug. 2010. URL: https://www.fvap.gov/uploads/FVAP/VSTL_AppendixB.pdf (visited on 07/01/2015).

[199] *Verasco*. URL: http://verasco.imag.fr/ (visited on 07/01/2015).

[200] *Verified Software Toolchain*. URL: http://vst.cs.princeton.edu/ (visited on 07/01/2015).

[201] *Verifying Multi-threaded Software with Spin*. URL: http://spinroot.com/ (visited on 07/01/2015).

[202] Kim Walden and Jean-Marc Nerson. *Seamless object-oriented software architecture: Analysis and design of reliable systems*. New York: Prentice Hall, 1995. ISBN: 0130313033.

[203] *Web Accessibility Evaluation Tool*. URL: http://wave.webaim.org/ (visited on 07/01/2015).

[204] *Web Accessibility Initiative*. URL: http://www.w3.org/WAI/ (visited on 07/01/2015).

[205] David A. Wheeler. "Fully Countering Trusting Trust through Diverse Double Compilation". PhD thesis. George Mason University, 2009. URL: http://www.dwheeler.com/trusting-trust/ (visited on 07/01/2015).

[206] *Xen Project*. URL: http://www.xenproject.org/ (visited on 07/01/2015).

[207] Xuejun Yang et al. "Finding and understanding bugs in C compilers". In: *ACM SIGPLAN Notices*. Vol. 46. 6. ACM. 2011, pp. 283–294.

[208] Shin Yoo and Mark Harman. "Regression testing minimization, selection and prioritization: a survey". In: *Software Testing, Verification and Reliability* 22.2 (2012), pp. 67–120.

[209] *YouTrack*. URL: https://www.jetbrains.com/youtrack/ (visited on 07/01/2015).

[210] Filip Zagórski et al. "Remotegrity: Design and Use of an End-to-End Verifiable Remote Voting System". English. In: *Applied Cryptography and Network Security*. Ed. by Michael Jacobson et al. Vol. 7954. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 441–457. ISBN: 978-3-642-38979-5. DOI: 10.1007/978-3-642-38980-1_28.

[211] Daniel M. Zimmerman and Rinkesh Nagmoti. "JMLUnit: The Next Generation". In: *International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2010)*. Paris, France, June 2010.

[212] Philip Zimmermann. *The Official PGP User's Guide*. Cambridge, MA: The MIT Press, 1995. ISBN: 0-262-74017-6.