The E2E-VIV Report

Many People

June 22, 2015

Contents

Note: Names following chapter titles are the currently-assigned writers; percentages following writer names are very rough estimates of the approximate percentage of completion. Some material factored into the percentages may not yet appear in the generated report because it needs to be brought in from external sources.

List of To Do Items

Chapter 1

Executive Summary (Joe K./Susan) (100%)

Internet voting was first proposed over thirty years ago. In the intervening years, a handful of companies and a twenty-odd governments have created Internet voting technology that has been used in thousands of elections to collect millions of votes all around the world except, until very recently, in the U.S.A.

Every month, while one government experiments with a new computing technology into their elections, another decides that their last experiment was a failure and they shut down yet another very expensive failed IT project. All the while, a dedicated group of scientists and activists—hackers, cryptographers, cyber-security geeks, usability experts—have fought tooth-and-nail against the use of insecure voting systems around the world.

Their fight, and those governments' excitement, is particularly focused on Internet voting. After all, voter turnout is at a seventy year low, many millennials are politically disengaged, and voting equipment purchased with HAVA funds is falling apart and there is little state and federal funding to replace it. This is the a perfect storm for our democracy, and Internet voting seems to be the silver bullet.

After all, the Internet permeates our lives. Smart phones. Social media. Online banking. Streaming music and movies. Google. Bitcoin. Online gaming. Why not voting?

Imagine a world where inexpensive elections have high participation rates by a well-informed, engaged public. Imagine elections where the disabled and abled have equal vote and equal opportunity. Imagine elections where corrupt electoral authorities or governments have no ability to manipulate the outcome. Imagine elections that truly capture the voice of the people and increase their confidence and trust in their government. Many imagine Internet voting as the solution that takes us to this democratic utopia.

Then imagine a world where millions have voted, we are about to tabulate a hundred million digital votes, and suddenly the electoral authority discovers intruders in their servers.

Or worse yet, imagine that exit polls show that the outcome is razor thin between candidates "Alice Democrat" and "Bob Republican", those responsible for the election push the button to ask their election server who has won, and the machine says "Charlie Communist"' and hundreds of politicians, electoral officials, and reporters look at each other with wide-eyed disbelief.

Today's Internet voting technology guarantees these kinds of horrific failures. Digital activists self-identifying as Anonymous with a grudge, Super PAC-contracted blackhat hackers, and enemy nation states will use their nearly unlimited resources to ensure such.

The only possible solution to this digital democratic dilemma is End-to-End Verifiable Internet Voting, or E2E-VIV for short. How to create an E2E-VIV for the American public is the focus of the report you hold in your hands.

E2E-VIV systems hold the promise of secure, trustworthy elections. They do so by giving voters, elections officials, and the general public the ability to observe an election and determine that it is well-run, not manipulate, and has a trustworthy outcome.

E2E-VIV system provide these verifiability guarantees by permitting (1) voters to check that their vote was recorded correctly, (2) voters to check that their vote was included in the final tally, (3) election officials to determine if there was any election manipulation or errors and mitigate such, and (4) the general public to count the vote and double-check the announced outcome of the election.

Meanwhile, while providing all of these guarantees, voter privacy is preserved and vote selling and voter coercion are avoided and attempts at such can be detected.

In other words, we maintain the security and privacy of today's paper-based elections, and yet increase the trustworthiness of our elections.

While the promise of E2E-VIV is decades old, our ability to design and build such a system, especially in the face of enormous security threats, is only recently possible with recent advances in computer science and cyber-security. High-assurance software engineering, software formal verification, formal computer-assisted verification of advanced cryptographic protocols and algorithms—none of these were practical, and some where not even possible, as recent as ten years ago.

Internet voting must be not only end-to-end verifiable, but it must also be usable, accessible, and open source. As such, this report contains over one hundred requirements that must be satisfied by any E2E-VIV system.

Security and usability are always at odds. Nearly all E2E-VIV protocols designed to date focus on security at the expense of usability. Only recently have cryptographers started to consider usability as a primary requirement when designing new protocols. It is a recognized serious challenge that an future work in this area must address.

Software is only trusted by security professionals if its description, including all of its documentation and source code, is publicly peer-reviewed. This does not, of course, mean that anyone can change the code. It does mean that anyone can read and comment on it, improving its clarity, elegance, and correctness for the benefit of all.

This report contains the plan for executing on this difficult but worthwhile mission. The cryptographic, architectural, and engineering foundations on which this edifice must be built are spelled out in this report. How must cryptographic protocols be created, evaluated, and certified? What kinds of architectures facilitate deploying the system for different sorts of electoral authorities, including considerations for their certification constraints, finances, and existing electoral processes. What are the best practices in the design and development of mission-critical systems, as public elections are obviously nationally critical?

The four key recommendations made in this report are:

1. Public elections should not be conducted over the Internet using systems that are not end-to-end verifiable.
2. 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.
3. E2E-VIV 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.
4. E2E-VIV systems must be usable and accessible to the typical set of abled and disabled voters.

Living up to these recommendations, while fulfilling the requirements contained in this report, is an incredibly challenging, but is also an enormous opportunity to do good for the world. America is known for tackling the most difficult problems, as well as for striving to help other democratic countries. Moreover, America has most of the best minds and companies is the world that are capable of fulfilling this mission.

The future of End-to-End Verifiable Internet Voting is clear: we must move to a second phase of this project and tackle the recommendations head-on. We must create a usable, secure, correct prototype E2E-VIV system that fulfills the requirements contained in this report so that electoral authorities countrywide can begin to experiment with Internet voting, first for their UOCAVA and disabled voters, and perhaps later, only after those experiments are an unmitigated success, for the general public.

Now is the time. We must take this opportunity. The alternative is a future where Internet voting is a reality, but we have no confidence in our elections or democracy.

Chapter 2

Introduction (Joe K./Susan) (100%)

2.1 The E2E-VIV Project

Elections have been conducted for millennia, but the technologies used to cast and tally votes have varied and evolved tremendously over that time. In 2015, much of our discourse takes place online, and many call for elections to follow this trend and ask why they haven't done so already. One community that is especially desirous of a better approach is that of "overseas" voters, for whom voting often requires extraordinary effort.

In March 2013, Overseas Vote Foundation's President and CEO began a discussion with a small group of experienced election integrity technology advocates about how, if faced with having to specify an Internet voting system, they would respond. Despite their concerns about the security of Internet voting efforts to date, it was nonetheless agreed that taking on the question was crucial at the time.

Gridlock around Internet Voting is not unique to Washington politics. Across the U.S., the scientific community, federal agencies, cyber security specialists, and certain organized activists have strongly advised against exposing the ballots of the most powerful nation on earth to the seemingly endless range of cyber threats which run rampant on today's Internet.

Nevertheless, faced with ongoing challenges to serve their constituencies in modern and efficient ways, and having experienced the everyday efficiencies of technology throughout their lives, when seeking new and improved election systems, election officials often want to consider Internet-based technologies. In the current climate of economic austerity, innovation in elections is rare. Our election officials are trapped in a technology purgatory; they continue to devote scarce resources to support outdated voting systems, while lacking the means to certify the new voting systems they would like.

Election integrity advocates assert that secure, tested, certified remote voting systems that election officials envision are not available. The scientific community does not consider online ballot return systems secure, and no such systems had been certified at the time of this writing. As a result, email has become a common interim method for moving ballots online, although it does not provide any of the benefits that a secure, full-featured voting system would provide. Email is demonstrably insecure, yet election officials and voters regularly use it to transmit ballots because no viable alternatives are available. Examination of new and better ways to use technology to meet specific voting needs, such as those of remote overseas citizens, military members, and people with disabilities is needed.

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. and not face the resistance of the election integrity advocates. No agreement on how to proceed, a years-long history of mediocre attempts, ongoing animosity between stakeholder parties, and a general lack of research on the current questions are among the challenges of this situation.

2.1.1 A Better Technology

In 1981, security researcher David Chaum published an influential paper describing a variety of applications of the then-new technology known as *public-key encryption*. Within that publication, a single short paragraph suggested how public-key encryption might be used to anonymize a set of ballots in such a way that enables universal verification of the accuracy of their tally.

Over the succeeding decades, numerous researchers have published hundreds of papers describing and refining various systems that employ what have come to be known as *end-to-end verifiable* election technologies. These technologies allow voters to verify for themselves that their votes have been accurately recorded and simultaneously allow any observers to verify that all recorded votes have been accurately tallied.

Dozens of end-to-end verifiable (E2E-V) election systems have been designed for a variety of settings—in-person and remote, paper-based and electronic, simple majority and instant run-off, etc. While early designs were cumbersome and difficult to use, more recent E2E-V election systems are far more fluid and natural.

There is little debate regarding the benefits that end-to-end verifiability can bring to election systems, and early steps are being taken toward large-scale deployment of E2E-V elections technologies in the context of in-person voting systems. The question at hand is whether the benefits of E2E-V technologies can adequately mitigate the legitimate security concerns created by Internet voting.

2.1.2 Definition

The term *end-to-end* is often used casually, without a precise definition in mind. For the purposes of this study, E2E-verifiability is a property of an election system with important components: first, that voters can individually check that their ballots are cast and recorded as they intend; and second, that anyone can check that all of the recorded ballots have been accurately tallied.[1] While systems of this nature have been developed in the past, none have been broadly used or successfully commercialized; the E2E-VIV Project has made a concerted effort to be informed by these past efforts and build upon them as appropriate. Usability factors were also considered from the outset of the study to address the significant challenges faced by remote and disabled voters when using such systems. This study is intended to enable development efforts in E2E-verifiable systems that are viable with respect to security, auditability, and usability.

For those concerned with election integrity, there is a justifiably negative reflex in response to IV: it takes all the problems with current remote voting systems and adds to them all the problems and security vulnerabilities of the Internet. The E2E-VIV Project has sought to make the case that use of the Internet enables and facilitates the introduction of E2E-verifiability (E2E-V), and that the benefits of E2E-V may be able to adequately mitigate the vulnerabilities introduced by using the Internet.

No participant in this project discounts the concerns of voting over the Internet, nor is E2E-V viewed as a magic fix that makes the Internet secure. Nevertheless, we believe that E2E-V properties are quite relevant to IV, since these properties are achieved even when votes are cast on potentially untrusted devices like personal computers and transmitted over an untrusted medium such as the Internet. The E2E-VIV Project does not attempt to make the Internet secure. Instead, it examines how E2E-V negates many (although not all) of the risks of voting via the Internet while introducing substantial new benefits that are not found in currently deployed voting systems.

2.1.3 A Proposed Study and Objectives

Within this context, a project proposal was developed and written by Susan Dziedusyzcka-Suinat of the Overseas Vote Foundation (OVF) [2] the leading nonpartisan, nonprofit organization dedicated to overseas and military voter participation and one that has maintained an unwaivering commitment to the cause of election integrity. Known for its work in developing user-oriented voter services for the full range of overseas and military voter needs, OVF was

[1] Definition from Dr. Josh Benaloh, Senior Cryptographer at Microsoft Research.

[2] Since the start of the study, the Overseas Vote Foundation has been renamed as "Overseas Vote", an initiative of U.S. Vote Foundation, which has broadened its mission to include U.S. domestic and absentee voters.

ideally suited to conceive of, define and manage this project. Over the past decade, OVF was at the forefront of introducing software solutions for this voter base, from its launch of the first online state-by-state customized voter registration/absentee ballot wizard for overseas and military voters, the first online Federal Write-in Absentee Ballot with dynamic candidate lists, the first Hosted System Solutions for states or organizations who wanted to offer to better overseas and military voter services to the first Election Official Directory API. The market space that OVF opened a decade ago has become a profitable one for many vendors since.

On December 19, 2013, OVF announced the launch of this project as the End-to-End Verifiable Internet Voting: Specification and Feasibility Assessment Study (E2E-VIV Project). The project funding was provided by 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." The proposal envisioned a project to examine the future of voting and how it might be executed securely online; one that approached the question of Internet-voting from a research perspective, and that sought to fill in the gaps of the many open questions plaguing the discussion.

The stated aim was to examine whether an end-to-end verifiable Internet voting system can be built that would offer a viable and responsible alternative to current systems to support the voting rights of overseas voters. According to Joe Goldman, Director of the Democracy Fund, "The significance of this project will be in its ability to break open the conversation from its current stalemate and include all sides in a constructive project to openly examine and research what is really needed by voters and election officials, and to determine whether this form of voting can meet those needs and still guarantee security of the election. Equally important, it will identify potential tradeoffs and shortcomings that represent the diverse range of values we hold dear in our elections."

For this study, a unique team of experts in computer science, usability, and auditing together with a selection of local election officials from key counties around the U.S. was been assembled. The focus is to produce a system specification and set of testing scenarios, which if they meet the requirements for security, auditability, and usability, would then be placed in the public domain. At the same time, a strong effort was made to demonstrate that confidence in a voting system is built on a willingness to verify its security through testing and transparency.

There is a historical misunderstanding in the U.S. election community that the E2E-VIV Project has aimed to correct. Our country's foremost scientists are not against technology advancements, nor are they inherently at odds with the election officials who seek technology improvements to meet their administrative challenges. Instead, the U.S. scientific community has continued to be extremely skeptical of unproven claims of security regarding existing systems that are not publicly tested or vetted. This study aimed to break this impasse and reconcile these concerns. 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 (IV) can happen, it must be in a system that takes advantage of end-to-end verifiability and auditability.

2.1.4 Shared Goals

Election officials and scientists involved in elections share the same overall goals: that voting systems provide accurate results, protect the privacy of voters, are easily used by all voters, and are robust against both accidental and intentional disruptions. Additionally, it should be publically demonstrable that these properties are achieved.

This project showed that the scientific community cares deeply about addressing the needs and requests of election officials as they serve remote voters, that they have a great motivation to address these questions when given a constructive opportunity to do so, and that they would like to work together with election administrators to examine the possibilities in this realm. It also demonstrated that the scientific community needs to improve at understanding the practical aspects of administration of elections and in collaborating and communicating with election officials to develop technical solutions.

2.1.5 Project Goal

The goal of this project was to specify and define a system and its testing scenarios for an online voting method that can

The tense of the entire introduction seems to change here, from present (study is intended,

provide both security and confidence to voters that their selections are accurately recorded and counted. The premise is that E2E-V negates many, although not all, of the risks of voting via the Internet while introducing substantial new benefits that are not found in currently-deployed voting systems. The project presumed that E2E-V is a possible answer, or at least a step toward one, and explored how well it can meet the needs of many voters and election officials. Additionally, any shortcomings found with this approach would serve as a starting point for future work.

2.1.6 Additional Objectives

A secondary objective of this work was the presentation and discussion of the report with key stakeholders, integration of feedback, and seeking of broad acceptance of the report's processes and conclusions (see Section 2.2.6).

2.1.7 Deliverables

The main deliverable of the E2E-VIV Project was the development of a "whole product solution" specification (or simply "specification") for a trustworthy E2E-VIV election system.

This report presents a system specification for a secure E2E-VIV system, a set of testing specifications to demonstrate the security, and a set of guidelines for system usability, accessibility, and testing. Additional topics and analyses may be considered and discussed in the report, such as legal and administrative challenges, and ballot secrecy, privacy, and confidentiality.

2.1.8 A First Step

This project represented the first step in an examination of whether one day E2E-VIV might be possible. The plan was to examine the potential for an E2E-VIV remote voting system together with election officials, taking into close account their needs and the needs of disabled voters. If a system can one day be developed based on these principles, then this would be determined. A viable outcome of this study with respect to security, auditability, and usability would enable development efforts to ensue.

2.1.9 Success

Beyond the concrete outcomes of this project, the fact that it took a research and testing-based approach to a problem that had been "in stalemate mode" would, it was hoped, stimulate election development overall. The election industry has been operating in a traditional paradigm 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 be the production of a specification for a usable, secure E2E-verifiable remote voting technology, to identify its strengths and weaknesses, and to develop reasons to pursue or not pursue this approach to remote and/or disabled citizen voting.

However, from the beginning it was clear that if the project were to determine that current techniques were weak and should not be pursued, then this would also be an outcome with many useful implications. A complete success for this project would be to produce a specification that would be: 1) supported by the vast majority of the contributors, including the technical, usability, testing, and research teams; 2) endorsed by the vast majority of the advisory council; and 3) endorsed by the major stakeholders in elections administration as represented by the project's local election officials. Additionally, the E2E-VIV Project hoped to receive support and endorsement from many members of the electronic voting activism community, as represented by key members of the Election Verification Network, the Verified Voting Foundation and beyond. The specification was intended to fulfill the following requirements:

Independent Implementation The specification must be of sufficient detail and clarity that an implementation of the election system must be possible by an independent party without extensive dialog with participants in the project.

Independent Validation It must be possible for a moderately proficient IT expert to objectively determine, in a reasonable time frame with 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.

2.1.10 Scope

The original project was tightly limited to involve system specification and testing only. No system development was envisioned in Phase I beyond mock-ups to help test usability. However, this changed early on in the project when Dr. Joseph Kiniry of Galois, Inc. was engaged as the technical project manager and introduced to the project a new engineering team.

Significantly, Galois is a leader in the process of computing on data while it remains encrypted, and in the automated generation, validation and synthesis of high assurance cryptographic solutions. They excel in multiple areas of cryptographic implementation and secure-by-construction software, all of which can be applied to the challenge of developing secure and usable E2E-VIV voting. The relevance of Galois's work to the project is clear: applying cutting-edge computer science and mathematics to solve difficult technological problems is needed to solve the secure, verifiable election systems development challenge. Galois's management agreed to donate a significant portion of engineering time to the project in order to build "demonstrators" that would be used to prove the concepts of E2E-V and to further examine security and usability.

The Galois engineers developed a set of rigorous engineering artifacts, "demonstrators", fit for refinement into a working election system, and against which third parties can perform independent validation and verification. Galois defined demonstrators as technical artifacts from the point of view of definition and constructions, but non-technical artifacts from the point of view of demonstration. The demonstrators developed using Galois interent 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, and
- licensed under either a mainstream Open Source license with a strong community or an alternative license tuned to the elections community.

2.2 People

The E2E-VIV Project was 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 together to form the overall project team. Technical, usability, testing and local election official sub-teams were formed for ease of communication. The technical team had decades of experience in E2E-V technology, cryptography, usability, and testing. An Advisory Council was established to broaden the communication with interested members of the election community.

Overseas Vote Foundation (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. Deep experience in the arena of overseas and military voting, absentee voting, community building, voter survey research, election reform and communications gave OVF a unique edge in managing the project.

Galois, Inc. provided the technical and engineering project management. Named as the technical project manager, Dr. Joseph Kiniry, working as a Principal Investigator at Galois, facilitated the communication and decision-making of the team. He became the main author and editor of the report and ran all engineering projects and usability aspects of the study.

2.2.1 Team Members

Project Manager: Susan Dzieduszycka-Suinat, Overseas Vote Foundation

1: We'll be turning this section into a more attractive illustration to enumerate participants in the project. -JRK

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.

2.2.2 Stakeholder Groups

Although not on the official project team, there are several communities relevant to the E2E-VIV Project outside of those represented on the project team, and interaction with members of these communities has been essential. These communities include the following overlapping cohorts.

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

Their skeptical attitude is compounded by the fact that several vendors have developed Internet voting products which are proprietary, closed-source, have never seen a public audit, and are unverifiable. Moreover, many of these vendors make specious claims about the security of their products—claims which the advocate community rejects entirely. Finally, many vendors advocate outsourcing elections entirely to them—a condition that will never be acceptable to the advocate community, even for an end-to-end verifiable Internet voting system.

A small number of advocates are **for** verifiable Internet voting, a small number are adamantly **against** Internet voting of any kind, but the bulk of advocates are on-the-fence. That majority recognizes that there are significant scientific and engineering challenges in designing and developing an Internet voting system. Moreover, they recognize that the decision to deploy such a system is very much a subjective, political one. In some contexts, it is viewed as perfectly acceptable to use a non-verifiable, outsourced election apparatus (such as Everyone Counts' product); for example, in an election to decide the winner on a reality show. But for government elections of any value, such an option is unacceptable to virtually every advocate.

Consequently, being fully transparent with—and listening to the feedback from—the election verification advocate community is absolutely mandatory. If the bulk of that community is not swayed by the evidence presented in this report, pursuing any next phase in this project will be fraught with turmoil and will be an uphill battle against many influential actors, all with good intentions.

Standards Bodies. Perhaps surprisingly, there is little national or international standardization in the area of elections. A nascent effort to begin standardizing data interchange formats began about a decade ago and eventually fizzled after only producing one small standard.

There are a myriad of reasons why this first effort failed. Vendors lobby against, and are disinterested in, interoperability. The Election Assistance Commission's Voluntary Voting System Guidelines (VVSG) 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. Finally, there was insufficient acceptance from the election research community.

In 2015, this situation changed with the rebirth of the IEEE 1622 committee focusing on elections. The IEEE Voting System Standards Committee 1622 (VSSC/1622) is creating standards and guidelines around a common data format for election data. The aim is that future election equipment used in U.S. elections and abroad can interoperate more easily. It is the intention of the VSSC that standards and guidelines being developed will be required in future versions of the EAC's VVSG.

Many of the top researchers, election advocates, and election officials in the world are a part of this committee. Additionally, representatives from the major election systems vendors are either participating, or listening in, because they recognize that interoperability will be mandated by future versions of the VVSG.[3]

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 not unreasonable to expect that different subsystems will be created and supported by different organizations. Second, in order to effect 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, they must be certified by the EAC. As such, EAC, NIST, and IEEE standards must recognize their core capabilities, subsystems, interfaces, and what constitutes legitimate evidence of correctness and security. Moreover, it is sensible to presume that there should be a standard means by which these aspects and evidence are documented to expedite a sound, accurate, and expedited certification process.

Vendors. Vendors relevant to this project come in two forms: (1) existing vendors of proprietary election systems and (2) future vendors that support open source election systems.

Existing vendors are significantly 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.

Obviously those that have an existing non-E2E-V Internet voting products (e.g., Scytl, Dominion, and Everyone Counts) will benefit from this work, whether that attention is justified or not. Any increase in attention on—or any hint of support from the verifiable elections activist community about—Internet voting aligns with their marketing goals.

[3] Recall that all election systems in the U.S.A. must be certified at the State or Federal level according to the EAC's voting system testing and certification standards, standards which mandate compliance with the VVSG.

Additionally, vendors can use the requirements herein in good and bad ways. We are hopeful that they will actively and positively absorb the recommendations of this report and engage with any future phase of this work.

From an optimistic perspective, if vendors read and understand the report and its recommendations, then they can closely track the work of the (provisional) next phase of this project. Any investment they make towards making their commercial systems end-to-end secure and verifiable—so long as there is publicly available evidence of such improvements—would be welcome by the bulk of the research and activist communities.

The pessimist will argue that vendors will simply use these requirements as a checklist, making specious claims that their systems are E2E-V but providing no evidence of such.

Time will tell which of these paths each existing vendor will choose to take.

The other kind of vendor that is relevant to this project does not yet exist: vendors that support open source election systems in manifold ways. Within the open source ecospheres—like those surrounding operating systems like Linux, blog platforms like WordPress, and content management systems like Drupal—there exists a pluripotent variety of companies that support these platforms: value-added resellers, integrators, hosting facilities, hardened software stacks, etc. Most of the companies provide commercial support for open source products at a variety of service levels, either via a subscription service or using a release-based license model.

We expect that a similar flowering of commercial support organizations must come to exist to support any E2E-VIV system that results with this project as its genesis. It is important to note that these organizations need not become experts in the underlying cryptography, formal methods, or usability and accessibility. The high-assurance development method proposed in this report results in validation and verification artifacts that are evidence-generating for correctness and security properties. Additionally, modern formal system specification languages support traceability from requirements to evidence.

Consequently, any future mission-critical E2E-VIV system is robust to integration, customization, or evolution work by these different kinds of support vendors. That is to say, any future E2E-VIV system developed according to the requirements and recommendations herein cannot be accidentally or purposefully broken without third parties detecting the certification failure. It is only by virtue of having these complete, consistent, traceable, evidence-generating artifacts that such a blossoming of support vendors can come to exist.

Hackers and Hacktivists. Hackers and election hacktivists are an important audience for this project. On the one hand, hackers, constructively or destructively, help make open source systems more secure. Hacktivists, on the other hand, 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 (e.g., in The Netherlands). Within these subcommunities, it is an undisputed truth that the design and development of a secure system used for public good must be open source and must witness critique and improvements from the public. Leveraging that attention is especially valuable for nationally critical systems like public elections.

Consequently, it is our expectation that direct engagement with these cohorts, from the Chaos Computer Club to Anonymous, while potentially tempestuous at times, is wise and will have manifold benefits.

Election Officials. Local elections officials (LEOs) are the core stakeholders in the design and deployment of E2E-VIV technologies. As such, not only must we pay attention to our expert LEOs on the project, but we must solicit feedback and reflections from any election official, past or present, who can give voice to their jurisdictions, and its voters' needs.

There are over 10,000 election jurisdictions in the U.S.A.; and there are consequently over 10,000 different ways that elections are run. Smoothly integrating with existing election processes is important. Base issues like common data formats and API integration are simple technical challenges that have straightforward, though potentially politically delicate, solutions.

The enormous plurality of technologies, large and small, at the federal, state, and local level make for difficult deployment of any new election technology. It is clear that traditional IT practices for the design, development, and maintenance of election software cannot work in this setting. No vendor can support 10,000 forks of a single code base to support 10,000 clients with slightly different requirements. This state of affairs leads to some of our technical recommendations, particularly those that touch upon feature modeling and software product lines.

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

Artifacts like documentation and tutorials, webcasts and screencasts, reports and demonstration software only go so far. An open supportive community must exist around E2E-VIV. This community may be realized in several ways. In the Open Source world, it is common to see friendly online forums for newbies spring up and regularly scheduled and community-organized developer and user conferences. Moreover, we expect to see a wellspring of companies ready and willing to support LEOs, ranging from value-added resellers to integration partners, as discussed above.[4]

As we exit Phase I of this project and, speculatively, move to Phase 2 - R&D, a wide range of elections officials must be actively engaged in the design and deployment of E2E-VIV technology, and we must foster the formation of a healthy, active, and supportive community, including volunteers and commercial IT firms, around it.

Voters. Despite the import of all of the aforementioned stakeholders, truly the most important stakeholder—in fact the one that holds the veto power over the wide-scale deployment of E2E-VIV technologies—is the voter.

It is clear that many citizens want flexible, comfortable, location agnostic voting. It is also clear that they do not realize the implications of this desire, particularly with regard to the security and privacy challenges of such a framing. Educating voters about the challenges of Internet voting is worthwhile, but it is clearly not possible to make every overseas and military voter aware of the utility of, and consequent critical need for, E2E-V election systems.

In fact, based upon our early usability studies of E2E-V election systems, it is not even clear that we can expect that a small fraction of voters understand and use the verifiability features of such systems, even if such systems see significant refinement and are truly usable by all voters.

Voters' trust in their election apparatus, and its transitive impact on their trust in their sitting government, is paramount. Consequently, changes in elections that may impact trustworthiness are difficult to make.

Voters are sensitive to changes that are bluntly visible, like e-pollbooks or e-voting machines. The DRM revolution and VVPAT redux have taught some voters to not trust technology at face-value.

Reactions to changes that have the potential to be misunderstood by, or cause alarm with, a vocal minority are unpredictable. Witness the introduction of voter ID across the U.S.A. over the past decade, as well as the alarm raised by the use of email ballot return, even in extraordinary circumstances like post-Hurricane Sandy elections.

Finally, and perhaps most critically, there is a delicate balance between comprehensibility and security—the Scylla and Charybdis of digital election systems. Paper-based elections are viewed as being transparent and comprehensible. Digital elections have lost those properties but, in the right circumstances, gain others like tabulation efficiency, decreases in residual vote count, and facilitation of independent voting for disabled voters. Internet voting again shifts the balance towards voter convenience, with a significant loss in general comprehensibility, since 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 thus based upon cryptographic principles, is a policy decision that firmly delegates trust and responsibility into the hands of a precious few. Those few—on the front lines, the elected official and the politician voting for modernization, and behind the scenes, the cryptographers, scientists, and engineers responsible for the system's design, development, validation, and verification—have enormous responsibility in their hands.

[4] For several examples of these communities, look no further than those that spring up around Linux distributions, Content Management Systems, blog platforms, wiki platforms, etc.

2.3 Methodology

In order to transparently and rigorously shoulder this burden in this, in the first phase of the E2E-VIV project, Galois organized the technical project management in a workflow that focused on delivering an evidence-based report and its complementary open source technical artifacts. This methodology is summarized as follows.

1. **Absorb all input from team members.**
2. **Read all literature on Internet voting.**
3. **Write baseline business and technical requirements and solicit feedback from technical team.**
4. **Write personas as foundation for UX studies.**
5. **Interview LEOs based upon requirements and personas; include in interview information about their current elections framing.**
6. **Outline report and solicit text and reflections from team members.**
7. **Reflect upon latest advances in cryptography for E2E-VIV.**
8. **Reflect upon latest advances for reasoning about cryptography algorithms, protocols, and implementations.**
9. **Craft a parameterized architecture space that reflect underlying requirements and standard cryptographic protocols.**
10. **Integrate all team member text and reflections and craft a final report table of contents.**
11. **Solicit more text and reflections from team members based upon final report table of contents.**
12. **Write chapters that were as-of-yet unwritten by team members.**
13. **Solicit input from all team members on Part 1.**
14. **Solicit input from technical team members on Part 2.**
15. **Solicit initial reflections from technical team members on feasibility of Part 2 potential recommendations.**
16. **Gather all input from team members (good and bad, nuances, disagreements, etc.) and capture all in appendices, citations, and footnotes.**
17. **Identify useful illustrations and figures for report and farm out drafting of such to illustrator.**
18. **Copy-edit, layout, polish, and release report.**
19. **Cleanup and make public the git 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.

2.4 Outcome

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 very 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 a parameterized architecture space in which solutions can be designed, built, and evaluated. Finally, the rigorous engineering necessary to create a high-assurance E2E-VIV product is spelled out via a recommended software engineering process, methodology, and set of technologies which, when used properly, can be used to fulfill the security and technical requirements while generating the necessary evidence that the system is fit-for-purpose.

Given these underlying recommendations, the assessment of the system by the expert team has had two possible outcomes:

1. **Positively:** The majority of the expert team may decide that the specified election system meets all of the requirements set forth by the charter of the group. This outcome would indicate that OVF might potentially move forward to ensure that the election system is developed and, potentially, deployed.
2. **Negatively:** The majority of the expert team may decide that the specified election system does not meet all of the requirements set forth by the charter of the group. This outcome indicates that further funding to design or construct such an election system is, for the moment, unwise and that the community believes that designing a usable and secure election system is still an open scientific, not engineering, challenge.

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

2.4.1 User Interface Design

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

At the moment, this report's system specification is focused on high-level requirements for any E2E-V election system (Internet voting-based or not). As such, a number of 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 is focused on developing a technical infrastructure and complementary process for the efficient definition and execution of qualitative and quantitative usability and accessibility studies.

The initial usability study that was conducted gathered qualitative feedback from several dozen voters. Based upon that study, whose full report is included in Appendix C, three conclusions were reached about voters' mental models about Internet voting.

1. *Voters intrinsically 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 disposition; 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 existing vendors to sell their technology and gather naïve voter feedback as "evidence" that their systems are fit-for-purpose for modern public elections.
2. *While many voters are supportive of having verifiability in their election system, very few voters are interested in learning about verifiability and taking the necessary time and energy to perform verification.* Fortunately, only a small fraction of voters needs to actually verify for most E2E-VIV election schemes to generate sufficient independent evidence that the election outcome is correct. As such, our requirements and recommendations reflect this level of voter engagement and suggest several means by which the threshold for election verification is always achieved.

2: Make sure requirements for verifiability and our recommendations are strengthened given the UX

3. *Voters expect the Internet voting experience to be somehow different from a traditional voting experience, be it on paper ballot or computer-assisted in a supervised setting.* It was not uncommon for voters to say, "Huh, that's it?!" after completing their voting process. Voters' mental models are trained to expect modern, rich web surfing experiences a la Facebook, YouTube, and Amazon and novel, interactive user experiences on their smart phones like Siri, Google Maps, and Uber. Consequently, this raised expectation opens the door for novel experimentation for a 21st century voter experiences.

Chapter 3

Remote Voting (Philip) (100%)

3.1 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.

3.1.1 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.

3.1.2 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 the exact number of "UOCAVA eligible" voters, but Figure 3.1 lists a recent estimate of the total.

3.1.3 Domestic Absentee

Domestic absentee voters are those who vote early in-person, or 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). 21,853,762 or 16.6% of all votes cast in the 2012 US general elections were from domestic absentee voters [1]. As of 2015, 27 states allow voters to apply for an absentee ballot without providing a justification, known as "no-excuse absentee voting".

3: Ask Philip for context on these numbers. Statewise breakdown doesn't seem to add much, and the difference between what the two are counting is unclear

State	**Overseas Voting Eligible Population** (McDonald 2009)	**Overseas military and federal civilian employees** (US Census Bureau 2010)
Texas	11.05%	11.78%
California	9.78%	8.44%
Florida	9.09%	9.54%
New York	5.31%	4.12%
Pennsylvania	4.10%	3.12%
Illinois	4.03%	3.24%
Ohio	3.51%	3.07%
Michigan	3.29%	2.68%
Georgia	2.84%	3.83%
Washington	2.78%	2.77%
North Carolina	2.78%	2.91%
Tennessee	2.57%	2.81%
Virginia	2.51%	3.52%
Estimated Total	4,972,217	1,042,523

Figure 3.1: Comparisons of American Overseas Population by State

4: Take out this figure too?

3.1.4 Expectations

In 1952, a study by the American Political Science Association defined ten voting rights necessary for members of the armed services [13]. 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 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.
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, leaving much 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.

State	Percent of Population
Colorado	71.4%
Arizona	65.9%
Montana	57.5%
Georgia	48.8%
Iowa	43.1%
California	39.8%
Hawaii	36%
North Dakota	28.8%
Florida	26.8%
Michigan	26.4%
Wyoming	26.2%
Maine	25.5%
Nebraska	25.4%
Idaho	24.3%
Ohio	22.4%
Wisconsin	21.4%
Vermont	20.4%

Figure 3.2: Votes Cast as Domestic Absentee 2012 General Election

3.2 History

3.2.1 Armed Forces Voting

Before the American Civil War, US 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. However, in 1864, with the Civil War displacing soldiers from their residences, 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 absentee military voter laws were treated as temporary and repealed after the war.

For the 1918 midterm elections, the US War Department decided that it was not ready to support the military vote, going so far as to prohibit individual states from canvassing overseas soldiers serving in the First World War. The Second World War 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), which, although passed too late for the 1942 midterm elections, gave military personnel absentee voting rights for federal elections during times of war without subjugation to voting tax or postage costs. Nonetheless, the act was notable for mandating 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 [104].

3.2.2 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, but state laws were a confusing and inconsistent patchwork, lowering 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 Voting Assistance Act expanding the number of civilians covered by the law, though those amendments were once again voluntary recommendations to the states. As lobbying pressure increased further, the Overseas Citizens Voting Rights Act (OCVRA) passed in 1974 and for the first time guaranteed, rather than merely recommended, absentee voting rights for overseas civilians.

In 1986, the Uniformed and Overseas Citizens Absentee Voting Act (UOCAVA) [Note - UOCAVA was defined earlier in this chapter - see line 31] was passed, combining and replacing the FVAA and the OCVRA and making the rights recommended by the previous acts into requirements for both military and overseas civilian voters.

3.2.3 Disabled Civilian Voting

The Voting Rights Act (VRA) of 1965 was the first legislation to enfranchise voters with disabilities. The VRA granted voters who require assistance to vote by reason of blindness, disability, or inability to read or write, to receive the assistance by a person of the voter's choice.

The Voting Accessibility for the Elderly and Handicapped Act of 1984 (VAEHA) was passed to improve access for disabled and elderly individuals. However, as with the recommendations of FVAA and OCVRA, states were left to set their own standards of access, and limited the disabled voters group to those with *physical disabilities*. The VAEHA 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 American Disabilities Act (ADA) requires that people with disabilities have access to basic public services, including the right to vote. The 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."

The majority of federal laws passed to protect voting rights of disabled citizens have struggled to clearly define a representative range of disabilities, and are often focused on in-person access to physical polling locations, which is expensive for states to implement. Additionally, state-defined policies often ignore rights to voting privacy, and exclude persons with multiple disabilities for which assistive technologies are not yet available.

3.2.4 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, addressing some accessibility concerns.

HAVA is also a response to the large number of rejected ballots in the 2000 election and an inability to sufficiently audit ballots. HAVA recommends 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 federal Election Assistance Commission (EAC) to oversee the development of new voting machine standards, and released the Voluntary Voting System Guidelines to aid in this transition.

The Military and Overseas Voter Empowerment (MOVE) Act of 2009 addresses barriers to overseas voter participation, specifically attempting to reduce the number of ballots that were not counted due to late receipt. The MOVE Act requires states to transmit 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.

Historically, legislative progress on absentee voting moves slowly. Existing voter rights regulations are enforced at a state level and are hampered by local political attitudes. In 2010, the Uniform Law Commission, a nonprofit organization, 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.

3.3 Shortcomings of Current Practice

Despite years of progressively stronger legislation addressing the needs of remote voters, there are many shortcomings of current election practices. The topics listed below draw from specific concerns that have a large impact on remote voting participants.

3.3.1 Use of Communication Technologies

The majority of remote voting takes place via postal mail, which is subject to many inherent faults that are exacerbated for voters. The 2008 Post-Election UOCAVA Survey Report and Analysis found that 52% of attempted UOCAVA votes were not counted due to problems in the mail delivery process. Additionally, 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) for registration forms. Unfortunately, this is very large and difficult to follow. For states without streamlined online registration, there have been many failed voter registration attempts.

5: doesn't the Federal Postcard Application address this?

3.3.2 Accessibility and Usability

In 2007, 20% of Americans with disabilities said they were unable to vote in a presidential or congressional election due to barriers at or getting to the polls [99]. This is frequently a consequence of the voting technologies used and the physical location of polling places. In the 2000 presidential election, 56% of randomly sampled polling places in the United States had at least one identified impediments for disabled voters [83].

For in-person voting, disabled persons often forfeit privacy in order to receive assistance from an aide, often because polling locations have insufficient assistive technology, or because the assistive technology in place is too difficult for voters and poll workers to use. Remote voting still presents obstacles: those with dexterity impairments often have problems with handling and marking paper absentee ballots.

3.3.3 Auditing

Although it is believed that voter fraud is fairly uncommon, it is a major concern in a bipartisan system. It is very difficult to detect voter fraud without depressing turnout or disenfranchising legitimate voters. Policies intended to reduce fraud or protect identities, such as increasingly prevalent and strict voter ID requirements, often result in a higher rate of rejected ballots. This is the case even for remote voters. In the 2012 general election, over 20% of absentee ballots were rejected due to non-matching signatures or unmet ID requirements [1].

3.3.4 Voter Privacy

Voter privacy is a key expectation of a fair voting system. Privacy promotes voter independence, and helps prevent voter coercion and vote buying. Most remote voting practices require that voters forfeit independence or privacy due to the physical reality that election officials cannot enforce privacy in voters' locations outside the polling place. Several states require that voters sign a voter privacy waiver when casting a remote ballot [105].

Chapter 4

E2E-VIV Explained (Philip/Daniel/Adam) (100%)

6: signposting; some content is here, but we need an intro and transitions explaining what content is about to happen

4.1 Goals

Typical Internet voting election processes have six phases:

Setup During the setup phase, the election officials gather the information needed to run an election. This includes gathering registration information for all voters, identifying the issues and races that will be voted on, designing and specializing ballots, sending instructions and other information about the election to voters, and so on.

Distribution Once the election has been set up, election officials must distribute ballots to the voters. Different voting system architectures use different mechanisms, including postal mail, email, or by having voters interact with a website.[1]

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

Casting Completed ballots are then returned to the election officials; as with distribution, different architectures use different mechanisms.

Tallying The tallying phase includes the remainder of the election finalization tasks: counting votes and announcing the election outcome are common to almost every process, though some include other miscellaneous tasks like publishing certain 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 not!).

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 ballot that is received by and displayed to the voter match the ballot that was created and sent by the election officials; that the computer used to fill out the ballot faithfully reports the intention of the voter; and that the filled out ballot be received by the election officials exactly as it was when it was sent by the voter. Typical Internet communications involve not just the computers owned by the two parties communicating, but also many intermediary computers controlled by neither party. A good election system needs to account for this, making it impossible for these intermediates to intercept ballots for viewing or modification during transit.

[1] We distinguish between sending instructions to voters and distributing ballots; there is no hard and fast rule for the distinction, but a rule of thumb is that instructions are applicable to many voters, whereas anything that has been specialized for a single voter is part of the ballot and falls under the distribution phase.

Another concern is that voters computers are rarely administered by system administration professionals, and as a result many of them are compromised by outside forces. One consequence of this is that the voting phase itself may become corrupted: even if the ballot arrives unchanged to the voter, malware on the voter's computer 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 is resistant to this kind of attack without seriously sacrificing the usability of the system. Some systems use alternative distribution mechanisms as cross-checks; for example, sending a code to the voter by postal mail which can be used to check that the ballot displayed by their computer is correct.

To the extent that it is possible, it is desirable for Internet voting to be private and anonymous. Voters should feel comfortable voting they way they like (and not feeling pressured to vote for a particular candidate or to vote a particular way on some issue); the fewer people who know or can find out the way a given voter voted, the more comfortable they can feel. On the other hand, election officials only want to record votes from people who are registered to vote, and even then want to record only one vote from each voter. Thus there is a tension during the vote casting phase between retaining the anonymity of votes and ensuring that a vote is coming from somebody who ought to be able to vote. One popular approach to this problem in existing systems is to initially require each vote to be tied to the voter who cast it long enough to decide whether to include the vote in the later tally or not; then to keep the vote but delete the information about who cast the vote. This approach can work; however, audits of systems that take this approach show that it is all too easy to accidentally retain the connection between votes and voters longer than intended, and make this information much more widely visible than intended. From the privacy side of the tradeoff, it would be better if the voter could be confident that there was no connection stored because the information they send to the election officials during the casting phase does not include any personally identifying material.

7: ask Joe for a reference to the audit that showed that voting terminal logs retained exactly who voted for what

There is a subtle distinction being made here. We certainly want our Internet voting systems to be correct, private, secure, and so forth. It is important for the people developing these systems to verify that they are correct and take an active role in seeking out and eliminating defects in the system. But the goal of verifiable Internet voting is to go even farther: not just correct, but *visibly* correct. That is, it must be possible for the parties using the system to be able to *check* that the system is behaving correctly, without trusting in the abilities of the people who created the system to avoid bugs or trusting in the inability of third parties to influence the behavior of the system. As applied to anonymity: since it is not easy to prove to somebody else that you have deleted some information, one must simply avoid giving them that information in the first place.

This theme—of being not just correct, but verifiable—is one of the central ones of verifiable 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 crop up.

8: do we need some evidence of security vulnerabilities and cybercrime...?

The tallying process provides a particularly good example of the difference between correctness and verifiability. We 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. For example, some systems allow voters to check that their vote was included in the election outcome; some allow voters to check that the system is recording the content of their votes correctly; some even allow voters to check that the number of people that voted for a given candidate is accurately calculated without revealing any of the individual votes. Meeting these verification goals without violating the anonymity and privacy goals can be a balancing act.

Each of these individual goals contribute to a single top-level goal: end-to-end verifiability. The "end-to-end" property is that the whole election process produces a result that matches the intentions of the voters. The subgoals of this are summarized with the catchphrase, "Cast as intended; recorded as cast; and counted as recorded." This recapitulates the concerns discussed above; "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, and "counted as recorded" is the demand that the tallying process be faithful; and all of these demands are subject to not just correctness but verifiability, so that a voter can convince themselves that these properties hold even if they suspect that the system or election officials have been corrupted.

4.2 Shortcomings and Expectations of E2E-VIV

As discussed in Chapter 3, there are several difficulties with current voting processes: voters with disabilities cannot vote unassisted, communication channels with remote voters are slow and unreliable, vote tallying is labor-intensive and error-prone, and election audits are costly. Additionally, there is little visibility into the election process, meaning that individual 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 so on, to participate in the election. Internet communications are traditionally speedy (seconds per message 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 there is some hope that verifiable systems can make elections more transparent for this purpose, too.

There are also some serious challenges in rolling out an Internet voting system. Chapter 7 discusses the feasibility of producing a system that meets the security and verifiability goals we have touched on above. In addition to those concerns, the ability of normal voters to use the system to cast their vote in the way they intend to vote is a major goal; as we discuss below, current systems do not meet this goal very well. One component of this is the system itself; though another that is common to all Internet voting systems is the need for voters to have Internet access. This is not necessarily possible for all overseas and military voters.

9: would be nice to have some hard data about this

4.3 E2E-VIV in Practice

A number of practical voting systems have been developed based on the principles of E2E-VIV. This section describes several systems that have been used in a real election or in a pilot.

4.3.1 RIES

RIES, the Rijnland Internet Election System [62], was first used in 2004 to support elections to the Rijnland water management board, supplementing the system of postal voting used by the water board. A subsequent version was used to allow expatriate voters to participate in the Dutch parliamentary elections [59].

Before a RIES election, credentials are mailed to every voter in the form of a very long number. The same mailing also includes instructions for the voter.

During the election, voters log into an election web site that includes a client-side voting application written in JavaScript. The client-side application encrypts the vote by passing the voter authorization code and the public ID of the candidate through a one-way function to create the encrypted vote. The encrypted vote is then placed on a public 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 with all candidate IDs.

The algorithms and protocols used by RIES are public, and each voter, having access to all of the inputs and outputs, may (in principle) check the computations. This is weaker than the desired individual verifiability, but nonetheless, far stronger than conventional voting systems.

The Organization for Security and Co-operation in Europe (OSCE) sent an election assessment team to observe the use of RIES in 2006. Their report contains observations of critical security features of the system that could not be observed [88]. Further weaknesses were revealed by the Eindhoven Institute for the Protection of Systems and Information (EiPSI) in 2008 [63], notably that:

- the procedure of voter self-check is quite complicated,
- the two-channel (mail and Internet) voting makes 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
- there are realistic ways to forge votes via cryptographic hash collisions.

One of the more important lessons learned through RIES is that when voter authorizations are distributed long in advance of the election, a mechanism must be provided allowing voters to obtain replacement credentials and invalidate lost credentials. The 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 rife with 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 in principle can increase confidence in the election integrity, in practice it opens the system to spoofing and denial of service attacks. Furthermore in the RIES implementation the system is aware of the fact that it is processing a testing ballot, and all of the test ballots were voted identically from the same computer, limiting the confidence added at the expense of these vulnerabilities.

In the wake of these critical reports, plans to use RIES in the 2008 Dutch parliamentary elections were scrapped, and Internet voting as a whole was banned in the Netherlands.

4.3.2 Prêt à Voter

The state of Victoria in Australia held a governmental election in November 2014, using a version of the Prêt à Voter system [21, 28]. An attempt was also made to use Prêt à Voter in a student election at the University of Surrey in February 2007 [17]. The failure of this attempt illustrates 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 with the candidate names on one part and the voting targets plus 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 appears to be random. The voter marks her choice next to the candidate name of her choice, separates the two parts of the ballot, and destroys the candidate names. She may take a copy of the voted part home for later verification.

For tabulation, there is a cryptographically secure mapping from the ballot ID numbers to the apparent random order of the candidate voting positions. Multiple custodians using a mixnet or similar technique use this mapping to decode cast ballots into anonymized plain-text ballots which are then posted to a 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 the plain-text decoded ballots are easily counted for verification by any interested party.

An individual voter may also search for their voted ballot ID on the bulletin board. This reveals the positions that were marked on that ballot, but crucially, it does not show the corresponding candidate names. The voter may therefore verify that the positions marked at the polling place were correctly recorded by the election officials, but because the voter no longer has the part of the ballot linking candidate names to ballot positions, the voter cannot prove to anyone else how the ballot was voted.

4.3.3 Punchscan

Punchscan [93, 94] was used for the graduate student association elections of the University of Ottawa in 2007 [46]. 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. The system uses a two-part paper ballot where the top part has candidate names and candidate numbers (or letters) and the bottom part has numbered (or lettered) voting targets. Holes punched in the top part expose the voting targets below. The order of the voting targets for each race appears random to the voter. Both halves of the ballot bear an identical serial number.

The voter casts their vote by marking her choice with a bingo dauber, and the two halves are separated. Either side can be scanned (since the bingo dauber marked both through the hole and around it) as the cast ballot. The other side is destroyed, and a copy of the cast side may be retained by the voter.

A curious voter may inspect the public record of any cast ballot exactly as with Prêt à Voter. It does not matter which half of the ballot the voter retained; neither half contains the information necessary to determine the vote. Again, individual ballots may be audited, and the key to tabulating the votes is that there is a cryptographically secure mapping from the ballot serial numbers to the apparent random order of the candidate voting positions.

4.3.4 Scantegrity II

Scantegrity II (Invisible Ink) [30, 31] was used in the Takoma Park, Maryland municipal elections in 2009 [24]. In 2011, it was used for in-person voting with Remotegrity (Section 4.3.5) used for absentee voting. The 2009 Takoma Park election was the first use of an E2E system with ballot privacy in binding governmental elections.

Before the election, officials generate the seed to a pseudorandom number generator using a secret sharing scheme. Three-letter alphanumeric codes are created for each choice on each printed ballot using this seed, and additional tables are created so that interested parties can later confirm that the tally was computed correctly.

During the election, the voter experience is nearly identical to that of conventional optical-scan paper ballots. When the voter marks their 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 bulletin board.

After the election, public verification of the final tally proceeds with the public bulletin board in a manner similar to that of Punchscan and Prêt à Voter.

In addition to the public verification, an individual voter who takes note of their ballot ID number and the code revealed from invisible ink may use the public bulletin board to check that their ballot was indeed tabulated, though this information is not sufficient to prove that they voted a particular way.

4.3.5 Remotegrity

Remotegrity [130] is a remote coded voting system that was used for absentee voting alongside Scantegrity (Section 4.3.4) for in-person voting for the 2011 Takoma Park, Maryland municipal elections.

Remotegrity voters receive a coded voting ballot and an authentication card in the mail. The codes on the ballot can be covered by a lottery-style scratch-off field. The authentication card contains several authentication codes under scratch-off, a lock-in code under scratch-off, and an acknowledgment code. Both cards have serial numbers. The voter can be sent two ballots so that she can use one for auditing purposes.

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

She 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 the codes. This indicates to her that the election officials received valid codes for her ballot.

She 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 of 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.

As with RIES, if we assume that there is no communication between the computer used to print the credentials and the computer used to collect the votes, the latter computer does not know the mapping from codes to candidates, so the vote is not revealed to the computer. Further, because the computer does not know a valid code corresponding to another candidate on the ballot, it cannot change the vote. Finally, and uniquely, because the computer does not know the acknowledgment code, its presence on the election website assures the voter that the election officials received a valid code for her ballot.

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

If a jurisdiction is nervous 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.

4.3.6 Helios

Helios [6, 7] is a system developed for web-based Internet voting. It was used for the election of a Belgian university president in March 2009 and by numerous universities and associations since then, including the Association for Computing Machinery and the International Association for Cryptologic Research.

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

During the election, the voter enters their choices on the website. After entering her choices, the voter has an option to spoil their ballot in order to verify that it was recorded correctly. Upon completing a non-spoiled ballot, the system sends an email confirming the receipt of their vote, though not their choices. 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 homomorphic vote tallying with the optional addition of mixers and mixnets in some derivatives [20, 119].

Voter authentication is not required until after the voter decides to cast the ballot, so any interested party may prepare and audit ballots. All cast ballots are posted in encrypted form on a public 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.

4.3.7 Norwegian System

Between 2011 and 2014, the Norwegian government ran an Internet remote voting trial [55] using a cryptographic protocol designed by Scytl, a commercial voting system vendor.

The Norwegian system uses a three-channel model involving 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; only the last ballot cast is counted, and 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 end-to-end proof via an SMS 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 to protect 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." 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 system evolved significantly between its first use in 2011 and 2013, with added complexity to 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 systems in general as well as the particulars of the Norwegian system [25].

Scytl and the Norwegian government assert that this is an E2E system. However, the paper says that, if the voter's encryption software colludes with the return code generator (both presumably run on software written by Scytl), it can lead the voter into believing her vote was cast accurately even when it is not. Thus its property of voter-verifiability relies on voting system software behaving honestly. Additionally, the system does not provide a proof of the tally and hence the tally is not universally verifiable. Finally, there is no information that enables a voter to know which of her votes was counted (if she casts multiple vote to ward off coercion). Because the correctness of the tally is not possible without the correct votes being 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, Scytl as used in the Norwegian elections is not considered an E2E system.

4.3.8 Wombat

The Wombat voting system [61] has been used for multiple pilot elections in Israel. It is an in-person voting system where the voter votes on a touch-screen and obtains a printout of her vote with an encryption of it. The voter can choose to cast or audit the encrypted vote. If she chooses to audit the vote, she may check if the vote was correctly encrypted. If she chooses to cast it, the ciphertext is posted online, and she casts the unencrypted vote in the ballot box (this may be manually counted) and takes the ciphertext home. The votes are tallied using a verifiable mixnet.

4.3.9 DEMOS

DEMOS [40] is a coded vote system where the voter is given a two-part coded ballot; she audits one part and uses the other to vote. Associated with each choice on the ballot is a vote code—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 then make her choices. Note that if the voter scans the QR code, the scanning computer knows how she voted. The vote codes represent homomorphic encryptions of the votes and the verifiable tally is obtained in a standard manner.

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

4.4 Limitations of Existing 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.

10: reference other sections about this

Existing E2E systems mitigate some of these limitations. For example, code voting limits the ability for attacks against postal mail systems to change the candidates marked on voted ballots. However if an attacker simply intercepts and destroys the voted ballot, a replacement might not arrive in time for that voter to participate in the election. To mitigate this, election officials might choose to instead accept voted ballots via fax, email, or website, but such expedient measures often trade off the verifiability that makes an E2E system desirable in the first place.

In this section, we examine the limitations of E2E systems with a particular focus on the limitations that are unique to or exacerbated by E2E characteristics.

4.4.1 Voter Secrecy

Systems like Prêt à Voter (Section 4.3.2) and Punchscan (Section 4.3.3) rely on a randomized candidate order or a code on printed ballots to ensure voter secrecy. Voted ballots must appear on a public bulletin board in order to verify the election results, and so 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 and therefore identify which candidate is marked on the voted ballots, thus violating secrecy [21].

Recent writing on Prêt à Voter recommends printing ballots on demand at polling places in order to limit this possibility [100]. Printing on demand introduces additional problems and expense compared to centralized printing. More printing equipment is required at each polling place, that equipment can break or be difficult to operate, and the printing equipment must have some way of communicating with the rest of the election infrastructure to ensure it has, for example, the correct cryptographic seeds for generating new ballots.

Scantegrity II (Section 4.3.4) uses invisible ink to hide the vote codes on unvoted ballots, and Remotegrity (Section 4.3.5) 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 still can depend on voters and poll workers correctly following procedures. A voter can leave the polling place with a complete Prêt à Voter ballot, for example, failing to shred the half with the candidate order. With both halves of their ballot, they can prove how they voted, losing receipt-freedom.

RIES (Section 4.3.1) makes a deliberate secrecy tradeoff by weakening the receipt-freeness requirement in exchange for providing universal verifiability and a degree of individual verifiability. The results of an entire election can be independently audited with only the information 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.

4.4.2 Ballot Stuffing

As when ensuring voter 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 is not an inherently verifiable property of the system, and requires trust in the accuracy of the poll workers.

In the Helios system, officials can enter voters by email address, and so there is limited protection against insider ballot stuffing. Helios relies on individual voters verifying their votes. While an interested party may verify the tally for the entire election by checking that the collection of encrypted votes is counted correctly, there is no provision for the interested party to determine if votes were fraudulently cast on behalf of voters who had not voted [102].

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 have an (anonymized) provenance that can be traced back to before the election began and presumably 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 providing for late or same-day voter registration.

4.4.3 Dispute Resolution

Note that E2E systems provide voter verifiability: they enable a voter to determine if her vote was accurately recorded. However, in the instance when the vote is not accurately recorded, not all E2E systems provide the voter with evidence that she may use to convince an independent party of the problem. This presents a loophole that may be exploited by dishonest voters, who may claim that their vote is not accurately recorded when it is, calling into question an honest election.

An E2E system with the dispute resolution property enables a voter to present evidence to support claims of election fraud. As a result, it enables a third party to resolve a dispute between a voter (claiming her vote is inaccurately recorded) and the voting system (claiming it is accurate).

All cryptographic protocols in the literature that provide dispute resolution require either paper or a second channel (such as a smartphone) in addition to the machine the voter votes from, whether in a polling booth or remotely. (We consider only protocols intended for use by voters voting from machines that are not trusted).

This implies that any existing voting system with dispute resolution needs the use of a second electronic channel or paper.

4.4.4 Infrastructure & Equipment

Election equipment fails in practice. An E2E system must be resilient to failures while not giving up E2E properties. A system that lacks robust fallback mechanisms is not itself robust, but is only as strong as its weakest fallback. For example, if a remote voting website fails and election officials resort to accepting voted ballots by email, E2E guarantees are lost for all of 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 ballots to be printed on demand, a high-quality shredder for two-part ballots, and more sophisticated assistive devices. This complexity incurs additional cost and poll worker training requirements.

Many E2E systems post encrypted ballots during an election to a public bulletin board. In order to update the bulletin board in real time, these election systems are distributed systems, networked via traditional means or via a manual air gap. Depending on the networking scheme this can open equipment to distributed denial of service (DDoS) attacks, network partitions, inconsistency, and other problems inherent to distributed systems.

Internet systems compound the difficulties of distributed systems by requiring the systems to be accessible via the public Internet, increasing the possibilities for DDoS and other malicious attacks. Furthermore, many systems allow voters to use their own computers to vote, leading to pitfalls inherent when election officials lack control over the voting environment. Malware on the voter's computer might undermine security, incompatibilites might arise due to operating systems or web browser versions, and the network infrastructure between the voter and the central election system might be compromised with a man-in-the-middle attack.

4.4.5 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 more steps and complexity, presenting usability complications that have not been fully studied. The mechanics of marking a ballot become more complex with code voting as in Remotegrity, and position or shape matching as in Prêt à Voter and Punchscan. Individual verification, not even possible in traditional systems, is an entirely new process that voters must master to take full advantage of E2E guarantees. Many E2E systems have attempted to reduce the additional effort for voters who are not interested in participating in election verifiability.

In 2014, a team of researchers from Rice University undertook a quantitative, experimental study of the usability of Helios, Prêt à Voter, and Scantegrity II [4]. They aimed to quantify usability using the ISO 9241-11 standard axes of effectiveness, efficiency, and satisfaction. Their results show that these systems broadly fail on these axes even for typical voters who are uninterested in performing additional verification steps.

The Rice study found the systems were not effective as significant number of voters failed to cast a ballot with each system. Troublingly, many of those voters thought they had in fact successfully cast a ballot; in a real election they would have left the voting process unfinished without even knowing to ask a poll worker for assistance. By contrast, traditional systems have near-100% success rates [22].

The systems also lacked efficiency, as they all required significantly more time – almost twice as long – to complete as a traditional system.

The usability of an election system is crucial for that system to not disenfranchise voters, and for voters to generally have confidence in the election results. The Rice study shows that adding E2E guarantees can be a Pyhrric victory when the resulting system is unusable for non-expert voters.

The results of the Rice study have been challenged by McBurnett et al [75].

4.4.6 Accessibility

There are ability requirements for many E2E systems in various stages of the voting process. For example, a sighted voter is able to see the correspondence between candidate position and marking position on a Punchscan ballot, but a non-sighted voter cannot without assistance. In addition to obstacles to marking a ballot, some schemes with individual verification lack provisions for disabled voters to participate in individual verification without assistance. Information required for verification is frequently delivered through a paper receipt, an invisible ink code, or requires writing down receipt data.

Accessible verification protocols have been proposed that take care to protect voter secrecy and allow participation in individual verification [29]. However, these protocols require using accessibility equipment with an audio, sip-puff, or switch interface to read and mark the unencrypted ballot. The device must therefore be trusted not to record the votes, which would violate voter secrecy. The device must also represent the ballot faithfully to the voter so that votes are recorded as intended.

Requiring trust in assistive devices is not unique to E2E systems [99]. In non-E2E systems, though, voters are required to trust many aspects of the election. In the context of having to trust the chain of custody of ballots, the integrity of poll workers, and the outcomes of any audits, having to trust an assistive 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 have confidence in the results of an election. The additional requirement of trusting an assistive device is not a flaw of the E2E system, but, instead, a limitation inherent to the use of assistive devices. It may be mitigated by enabling the voter with the need for assistance to use multiple assistive devices, and by requiring that all voters, whether in need of assistance or not, use similar assistive devices or a universal interface. This ensures that the assistive device and/or interface is tested by multiple users.

4.4.7 Social & Political

Novel election systems face a difficult bootstrapping problem: in order to be adopted in large-scale elections, they must have a successful track record. However in order to build up that track record, systems must be successful despite the limited resources available during small-scale pilot programs. With limited resources, corners are cut in the implementation of the election system leading to a greater chance that problems with equipment, software, and support will undermine confidence in the system.

This confidence in election systems generally, and E2E systems in particular, is fragile in the eyes of the public. When election systems fail during an election or are revealed to have substantial integrity issues, the perception of all similar systems is tainted, no matter the differences between specific systems or the reassurance of E2E guarantees. Failure of a legacy computerized system can poison the well and make the public reject a novel system by association.

For example, The Federal Constitutional Court of Germany issued a decision in 2009 in the wake of a hacking demonstration on electronic voting machines used in previous elections [53]. They 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", a standard which E2E systems have not been able to meet in practice [22]. Similarly after reports critical of RIES, a popular movement successfully advocated for a ban on Internet voting in the Netherlands.

Broader computer security concerns are becoming topics of household conversation with vulnerabilities like Heartbleed and droves of personal data compromises making the headlines. These concerns rightly make the public 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 accessible to non-experts without making it more difficult to vote.

Chapter 5

Required Properties of E2E Systems (Dan) (100%)

In August 2010, the U.S. Election Assistance Commission issued a set of testing requirements for UOCAVA remote electronic voting system pilot projects [121]. The general categories of requirements specified by the EAC included functional requirements, such as the need for the system to produce paper records of voter choices and generate human-readable ballot images; requirements on software development, such as allowable programming languages and coding conventions; usability, accessibility and privacy requirements, such as that a voter's ballot choices must remain private and that provisions must be made to support voters with disabilities; security requirements, including logging requirements, requirements on communications security within the system, and requirements on physical security and penetration resistance; quality assurance requirements describing the testing that must be done on the systems; and requirements about configuration management mechanisms, technical information, and documentation to be provided by system vendors.

The EAC requirements have some serious shortcomings, one of which is that several of the requirements seem arbitrary. For example, they specify (in Section 2.1 of the requirements document) that the voting system shall achieve a target error rate of no more than one in 10,000,000 ballot positions, with a maximum acceptable error rate in the test process of one in 500,000 ballot positions, without any justification for those numbers. They further specify (in Sections 2.1.1.1–2) that "memory hardware, such as semiconductor devices and magnetic storage media, shall be accurate" and that "the design of equipment in all voting systems shall provide for protection against mechanical, thermal, and electromagnetic stresses that impact voting system accuracy" without any guidance on how to evaluate such accuracy or protective ability.

In addition to these shortcomings, some of the EAC requirements are inappropriate or invalid. The most obvious example of this is the set of requirements that mandate specific "structured programming" characteristics of software implementation languages (Sections 4.1 and 4.4), which seem to eliminate functional programming languages such as Haskell and Erlang—widely used in implementing high-assurance systems—from consideration entirely.

If these issues were addressed, the EAC requirements could serve as a solid baseline set of requirements for remote electronic voting systems; effectively, addressing the "IV" in "E2E-VIV". However, they are not strong enough to guarantee end-to-end verifiability, which—as previously discussed—is essential when considering Internet voting systems for use in real elections. Thus, we describe here a set of required properties for E2E-VIV systems that has significant overlap with the EAC requirements.

The set of E2E-VIV 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. Each of these groups is itself divided into several categories, and Figure 5.1 gives a high-level overview of these.

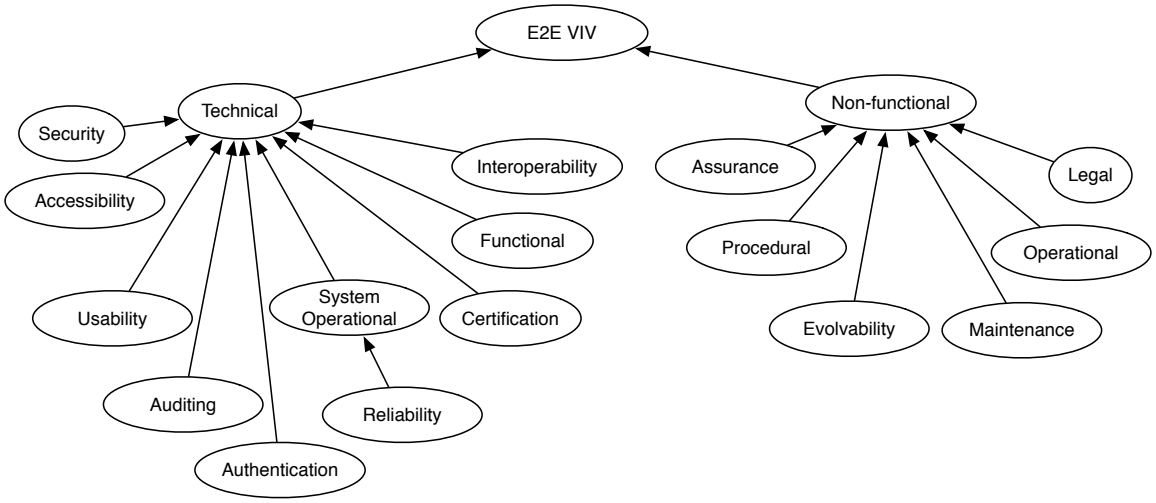

Figure 5.1: The hierarchy of requirements for E2E-VIV systems.

The following is a high-level description of the categories and many of the requirements within each; Appendix A contains a complete listing of all E2E-VIV system requirements expressed in the Business Object Notation.

5.1 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.

5.1.1 Functional

The functional requirements of an E2E-VIV system deal primarily with the casting and recording of ballots and associated voter records. One important requirement is that there must be a correspondence between the recorded ballots and the voters that are listed as having voted; a ballot cannot be recorded without a voter casting it, and a voter cannot be listed as having voted without casting a ballot. Similarly, if a voter is informed by the system that her ballot has been successfully cast, the system must correctly retain the record of her having voted and her cast ballot information even in the event of server failures.

Another functional requirement is the property of *receipt freedom*: it must be impossible for a voter to prove to anybody any information regarding how she voted her ballot, beyond what can be mathematically deduced from the final distribution of votes. For example, if a referendum passes with 100% of the vote, there is no way to hide the fact that every voter approved of the referendum; however, if the result is mixed, it must be impossible for any individual voter to prove how she voted. This must be the case 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.

Note that there is no such E2E protocol in the literature. In order for this requirement to be fulfilled, any protocol must allow the voter to vote multiple times. Yet, it must also allow it to appear that the video-recorded vote was cast, hence its encryption must be among those counted in a verifiable manner. Yet, this vote must not count, and the later vote should replace it. This represents a significant research challenge for cryptographic protocol designers.

In some elections voters are allowed to cast multiple ballots with only the last cast ballot counting toward the final election tally, while in others voters are prohibited from casting multiple ballots. The system must accommodate both of these election formats, ensuring that only the last cast ballot is counted for each voter when multiple ballots are allowed and ensuring that each voter casts at most one ballot otherwise.

Maintaining voter anonymity is critical, so 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 the correct earlier ballots. To balance these concerns, any link between a ballot and the voter who cast it must be irrevocably broken once it is conclusively determined that the ballot will be counted toward the final tally.

Finally, because the voter should be able to focus on the voting process without undue distractions or external influences, the voting system must not display or permit the display of any advertising or commercial logos during a voting session; the exception to this rule is that an election jurisdiction may display its own logo to the voter during the voting process. Along the same lines, the voting system must not display any links to other Internet sites outside of the voting system, except to provide help with the actual mechanics of voting.

5.1.2 Usability

The usability of an E2E-VIV system is critical to its successful adoption and use. Since the 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 one general usability requirement that applies to the system as a whole.

The first vote casting requirement is that, if a voter receives a final vote confirmation (e.g., "Thank you for voting!" or a similar notice) from the system, the ballot casting process is complete and the system has recorded the vote. This is the usability counterpart to the functional requirement that ballot records and voter records must be maintained correctly even in the event of server failures.

The second vote casting requirement is that, if a voter is uncertain whether or not her ballot was recorded (e.g., she clicked a "submit" button but never got a response from the system), she must be free to attempt to vote again.

Finally, usability testing must be performed 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 being deployed in a real election.

5.1.3 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 constraints at each stage of the development process. Consideration must be given to the system's compatibility with existing technologies designed to help individuals with disabilities; for example, the system should be developed in a way that allows assistive input devices such as switches, eye trackers and screen readers to be used in addition to keyboards, mice and touchscreens. Similarly, the system's presentation of voting options should be optimized to voters' needs by providing alternative display fonts, audio representations, braille representations, and other representations as appropriate.

All possible measures must be taken to ensure that the system can be used by all voters and, if that is not possible in all circumstances, to provide access to alternative methods of voting for those voters who cannot use the system.

Finally, accessibility testing must be performed in addition to the previously-mentioned mandatory usability testing. The reports of the accessibility testing must be made public, and the system must achieve satisfactory test results before being deployed in a real election.

5.1.4 Security and Authentication

Security and authentication are closely related and together represent the broadest set of technical requirements, consisting of both requirements on the E2E-VIV system itself (data storage, communications, etc.) and requirements on the voting and counting processes enabled by the system (voter authorization, voter privacy, tally accuracy, etc.).

It is crucial that data integrity be ensured throughout the system. Therefore, measures must be taken to ensure that no data can be permanently lost in the event of a breakdown or fault affecting the system; that the system maintains the integrity of the voters' register, lists of candidates, ballot information, cast ballots, and other critical information, in addition to authenticating the original source(s) of that information and tracking provenance where appropriate; that all data communications within the system have associated integrity checks; that system equipment under the control of the electoral authority is protected against influences that could modify the election results; and that the integrity of the election results does not depend in any way upon the security of system equipment not under control of the electoral authority. The system must perform regular "health checks" to ensure that data integrity has been maintained, that all its components are operating in accordance with their specifications, and that all system services are available.

Accurate timing information is critical to security, both in terms of providing evidence of compliance with applicable regulations and in terms of detecting 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 trails, election observation data, and time limits for various aspects of the election process. It must be possible to determine, using the timing information stored by the system, whether nominations (and, if required, acceptance thereof by the candidate or electoral authority), voter registration, and vote casting have occurred within the prescribed 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 is allowed by the legal requirements of the electoral jurisdiction. The system must allow access to each of its services only to authorized users; for example, only individuals who represent the electoral authority may be allowed to load ballot information into the system.

The authentication mechanisms 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. Authentication to the system may not be carried out through third parties; that is, existing online accounts such as those at Facebook, Google and Twitter may not be used as authentication mechanisms. The security of the authentication mechanism must not be affected by any potential breach of any public or commercial database (e.g., a credit card database, the Social Security database), and it should not be possible for an attacker 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 revokable at any time, at the behest of either the individual or election officials, and must be changed for all individuals at least once in every election cycle.

With respect to the actual voting process, only eligible voters may be allowed to cast ballots and the system must ensure that only the appropriate number of ballots is cast by each voter. It must be possible for a voter 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 privacy of the vote must be preserved end-to-end to the maximum extent possible, and individual voters may not waive the privacy of their votes. In the case of remote voting, vote privacy must be preserved even in the presence of arbitrary malicious code on the voter's computer (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 (e.g., Facebook, Twitter, etc.) regarding the act of voting. Any residual information that could be used to discover a voter's choices must be destroyed after a ballot has been cast; if a voter uses a computer outside the control of the electoral authority to cast her vote, she must be provided with instructions for destroying any such information on that computer.

With respect to vote counting, 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 *software independent*; it must be possible to reconstruct a correct tally from some record even if the election system software is compromised.

Finally, it is expected that a deployed E2E-VIV system will be an attractive target for highly-capable adversaries that wish to influence election results or to disrupt election processes. With this in mind, the system must be designed and tested assuming that an adversary has a budget of US$10 per voter per election that can be applied toward any critical subset of votes or voters of their choosing; thus, an E2E-VIV system for use in a U.S. presidential election would need to be designed and tested assuming that an adversary has a budget of approximately US$1,300,000,000.

The electoral authority shall have overall responsibility for compliance with these security requirements, and such compliance shall be assessed by independent bodies as appropriate.

5.1.5 Auditing

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

First, the audit system must be designed and implemented as part of the E2E-VIV system from the beginning; it cannot be added as an afterthought to an existing system. Audit and monitoring facilities must be integrated 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, and these logs must be unmodifiable once they are written and as complete as possible without violating voter privacy.

The audit system must actively report on 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. It 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, to detect voter fraud, and to 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 capable of giving an upper bound on the number of affected ballots. 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.

In order for an E2E-VIV system to be trusted, its auditability must extend to its own source code as well as the activities it performs during an election. Therefore, the E2E-VIV system software, including any official monitoring and auditing applications, must be published in source form along with documentation, instructions for building and running, and a digital signature as a proof of authenticity.

5.1.6 System Operational

System operational requirements ensure that the system is configured, updated, and run in a transparent, accountable way that allows for the other requirements to be fulfilled. One important such requirement is that there must be official published manifests of the system used to run any election, indicating details of the software and versions used, dates of installation, and brief descriptions of their functionality. Well-defined procedures must exist for both updating the manifests to reflect changes to the installed software and checking the installed software against the manifests to detect tampering.

Before every election period, all equipment (including all software) must be checked and approved in accordance with procedures devised by the electoral authority. This check must include a check of the software against the manifests, as well as any necessary tests to establish that the system complies with its technical specification.

During an election period, key equipment must be located in a guarded, secure area at all times. There must be a contingency plan for system failures including provisions for backup and failover systems, which must conform to the same standards and requirements as the systems they replace. In addition, sufficient arrangements for data backup must be in place, continuously monitored, and always available during the election; election staff must be ready to intervene rapidly, according to a procedure established by the electoral authority, in the event of incidents during an election. Individuals responsible for the voting equipment must follow established procedures to ensure that the equipment and its use satisfy requirements.

To ensure accountability on the part of the electoral authority and election system vendors, a report containing every software manifest change and every violation of data security, system security, physical security or control procedures must be prepared and made public by the electoral authority within a reasonable amount of time after every election.

5.1.7 Reliability

In order to be successfully used to conduct elections, an E2E-VIV system must satisfy strict reliability requirements with respect to both its behavior under normal conditions and its behavior while under attack.

In general, the back-end (i.e., non-voter-facing) components of the system must have a proven mean time before failure (MTBF) of at least one week under constant peak expected load; that is, it must have been shown in multiple actual tests of mock elections to run continuously for at least a week at the highest expected voter participation rate. The one week MTBF requirement applies only during normal operation, not while the system is under attack.

In addition to the MTBF requirement, the system must also exhibit 99.9% uptime during the election period, and 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, e.g., by unexpectedly unplugging servers or disconnecting storage devices. Redundant failover components must be in place for all critical components of the system in order to ensure the 10 minute maximum recovery time.

An E2E-VIV system is likely to be a tempting target for distributed denial of service (DDoS) attacks; it must be able to continue correct operation during a sustained DDoS attack at a specified level on any combination of its back-end components with no more than a specified acceptable degradation of response time to voters during the attack. The specified attack level and acceptable degradation of response time 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. Our 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 degradation of response time.

The ability of the system to survive DDoS attacks and continue operation while fulfilling the response time requirements must be demonstrated in the actual network configuration to be used during the election, and the required thresholds for these values should be re-evaluated every election cycle to keep pace with advancement in attack technology.

5.1.8 Interoperability

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

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

5.1.9 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 that demonstrate its fulfillment. These tests must be runnable on demand, and their results should be unambiguous and easily understandable.

In addition, the election protocol implemented by the system (communication, cryptographic, etc.) must have associated formal proofs of correctness and security to the extent possible. Note that until recent years it was rarely possible to provide proofs of integrity properties, hence security properties that are proven tend to be privacy properties, but this state of affairs is evolving rapidly in the verification community.

5.2 Non-functional Requirements

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

5.2.1 Operational

The operational requirements on E2E-VIV systems deal with several distinct issues including election and registration timing, voter registration, candidate nominations and lists, receipt freedom, voter assistance, and the handling of hardware and software platform issues and election integrity violations.

Voters must be informed, in clear and simple language, of how electronic voting will be organized and what steps a voter will need to take in order to participate and vote electronically. Support and guidance with respect to voting procedures must be available to all voters. In the case of remote voting, such support and guidance must be available through a different, widely-available communication channel (such as a dedicated phone number) in addition to being available via the Internet. Voters must receive clear guidance about exactly what client configurations (i.e., hardware platforms, operating systems, browsers, browser plugins, other applications, and versions thereof) are required by or supported by the E2E-VIV system, and what common components, plugins, or other software (e.g., pop-up blockers, script blockers) may interfere with voting. In addition, voters must receive clear guidance about configuration choices they can make to more strongly protect their privacy; for example, disabling cookies and browser history logging, running privacy-protecting browser plugins, voting from temporary virtual machines, logging out of social networks, disabling non-election-related Internet communications, etc.

In any election carried out using an E2E-VIV system, the relevant jurisdiction's legal provisions must provide for clear timetables concerning all stages of the election. The period during which a vote may be cast electronically must not begin before the public is notified of the election; in particular, with respect to 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, the time periods for remote and supervised voting need not be identical; however, remote voting should not be allowed after the period for supervised voting has ended.

An E2E-VIV system must have a publicly accessible voters' register that is regularly updated. Each voter must be able to check, at a minimum, that her information as recorded on the register is accurate, and must be able to request corrections of any inaccurate information. In jurisdictions where remote electronic voting takes place concurrently with voting at supervised polling stations, the system must be designed in a way such that it prevents any voter from voting more than once.

On any electronic ballot, all voting options must be presented equally; that is, there must be no distinguishing fonts, sizes, styles, or other embellishments that could cause one or more of the voting options to be perceived by a voter as "preferred". The ballot must be free of any information about the voting options—biographical information about candidates, interpretations of and statements about ballot initiatives, etc.—other than information strictly required for casting the vote or required by law to be on the ballot (for example, candidate party affiliation is often required to appear). The system must also avoid displaying any messages that may influence voters' choices. Additional information about voting options might be made available from an electronic voting site as part of an E2E-VIV system, separate from the actual electronic ballot; if so, such information must be presented without bias.

E2E-VIV systems are likely to be made available for testing by voters and election officials, both before and during elections. They must therefore indicate clearly, before the final casting of any ballot, whether the ballot is being cast in a real election or as part of a test. In the case of a test that occurs simultaneously with a real election, individuals casting test ballots should subsequently be directed to the appropriate voting channel for casting real ballots.

E2E-VIV systems must exhibit receipt freedom (mentioned previously in the technical requirements); that is, they must not enable the voter to possess a proof of the choices they have made in a cast vote. Receipt freedom has two different meanings, depending upon whether or not 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 should disappear from the display (visual, audio or tactile, depending on accessibility requirements) used by the voter to cast the vote as soon as the vote has been cast. When a paper proof of an electronic vote is provided to the voter at a polling station, the voter must not be allowed to show it to any other person or to remove it from the polling station. Note that the only existing protocols with receipt freedom are those that assume a private channel with the voting system and/or a trusted voting computer. This is not necessarily a reasonable assumption in a remote voting scenario.

In the unsupervised setting, as discussed in Section 5.1.1 above, the situation is different, though the underlying secure goal is the same. Even were an adversary/coercer were to digitally record the voting process or the voter were to record themselves with the intention of selling their vote, it must not be possible for the adversary to irrefutably conclude, either during the election or after the election is certified, that the coerced/sold vote is, in fact, as recorded.

With respect to counting the votes, an E2E-VIV system must not allow the disclosure of any vote counts until after the system has stopped accepting electronic ballots. Tally information must not be disclosed to the public until after the end of the voting period (including all polling station voting). Any decoding required for the counting of the votes shall be carried out as soon as practicable after the end of the voting period; representatives of the electoral authority must be able to participate in, and observers must be able to observe, the counting process. A record of the counting process must be kept, including timing information and identifying information for all persons involved in the counting process. In the event of any irregularity affecting the integrity of votes, it must be recorded that the affected votes had their integrity violated; the effect of such integrity violations on the election results will vary based on the legal provisions of the involved jurisdictions.

Finally, 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. The system must be auditable to the extent possible given this requirement, and the conclusions drawn from the audit process should be applied in future elections.

5.2.2 Procedural

Successful deployment of E2E-VIV systems requires certain procedures to be followed with respect to their provisioning, certification, maintenance, availability, and use. Because such systems are critical pieces of public infrastructure, information about their functioning must be publicly available and information about the specific components of a system must be disclosed, at least to the relevant electoral authority, as required for verification and certification purposes. Before any such system is introduced, at appropriate intervals after its introduction, and in particular when any changes are made to the system, an independent body appointed by the electoral authority must verify that the system is working correctly and that all necessary security measures have been taken.

After introducing a system, the electoral authority must take steps to ensure that voters undesrtand its use and have confidence in the system; these may include outreach, practice elections, and any other measures the electoral authority sees fit. In particular, voters must be given an opportunity to practice any new electronic ballot casting method before, and separately from, the casting of an electronic ballot during a real election.

The electoral authority must take steps to ensure the reliability and security of the E2E-VIV system; for example, guarding equipment, providing suitable reliable power supplies, etc. All possible steps should be taken to avoid the possibility of fraud or unauthorized intervention during the voting process, and the electoral authority must satisfy itself that the E2E-VIV system is genuine and operates correctly before using it to conduct a real election.

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

Observers must be allowed to be present, to the extent permitted by law, to observe and comment on the conduct and establishment of the results of any election conducted using an E2E-VIV system. During an election period, any authorized intervention affecting the system must be carried out by a team of at least two people, be the subject of a written report, and be monitored by representatives of the election authority 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.

Finally, there must be clear technical and legal procedures to be followed in the event that voters can prove that their votes were not received accurately or counted, or in the event that the official election verification application does not verify that the results of the Internet portion of the election are correct.

5.2.3 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, the voting interface of an E2E-VIV system must be understandable and easily usable, and registration requirements for electronic voting must not pose an impediment to voter participation. E2E-VIV systems should be designed, as far as is practicable, 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.

The E2E-VIV system must insure that at most one electronic vote from each voter is included in the final tally, that every vote cast electronically is counted, and that each vote cast electronically is counted only once. In jurisdictions where electronic and traditional voting channels are used in the same election, there must be a secure and reliable method to aggregate all votes, prevent multiple votes by the same voter from being counted, and calculate correct results.

The way in which voters are guided through the process of electronic voting should be designed to prevent 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, or to stop the voting process, without their previous choices being recorded or made available to any other person under any circumstances. The electronic voting system must not permit 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 (e.g., 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 such download and study, or for building, testing and publishing test results for the E2E-VIV system.

5.2.4 Assurance

There are several assurance requirements with respect 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. As previously discussed in Section 5.2.1, the specific supported platform and software stack combinations for the software must be clearly conveyed to voters. 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 E2EV core) should be published and available for free download by anyone. As the system changes, all associated 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.

5.2.5 Maintenance and Evolvability

Maintenance and evolvability requirements are closely related, and essentially stipulate that an electoral authority, or any entity engaged by an electoral authority, must be able to change an E2E-VIV system in response to changes in the legal or technical environment in which it operates.

The electoral authority 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. The electoral authority 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 and the procedural requirement that the system must be re-verified for correct operation before being used to conduct a real election.

Chapter 6

Crypto Specification(Aggelos, Joe) (100%)

As no existing E2E-VIV system, nor E2E protocol, fulfills the requirements set forth in this report, we cannot provide a full cryptographic system or protocol specification. The development and verification of such a specification will 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 system is a useful foundation for examining and comparing E2E protocols. Consequently, this chapter provides such a foundation.

This chapter also discusses the appropriate cutting-edge technologies that should be used to mechanize and verify E2E protocols and their cryptographic algorithms. Mechanization and formal verification is mandatory for this work. E2E protocols are of critical import for public elections, as national digitally supported elections are a critical system that directly impacts national security.

6.1 Crypto Specification

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, or in specialized environments such as EasyCrypt. The foundations discussed in the following sections has not yet been mechanized, but must be as a first step toward any future E2E protocol work. A discussion of the modern best-practices for such mechanization and verification is included in the conclusion of this chapter.

6.1.1 Ideal Functionality of an E2E System

In this section we introduce a template for expressing the core of an E2E verifiable 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 on such inputs. Furthermore, the ideal functionality specification includes an I/O interface with the adversary that 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 high-jack the communication of parties to and from 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. Contrary to that, in the "real world" such an ideal functionality does not exist and hence parties have to resort in 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 that no matter how the system is operated by the parties, it is impossible to achieve any distinguishing effect between the two worlds. The hallmark of a safe implementation of the ideal functionality is exactly this indistinguishability property. 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 such 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 of the environment with the ideal functionality). The latter property is essential to ensure the arbitrary composability of the election system within larger systems (that may involve other components such as deliberation platforms and so on). If the interface between environment and adversary is specified and restricted in some sense then it is typically the case that realizing the functionality becomes simpler however this may be done at the expense of sacrificing the composability of the protocol.

For our objective, we will provide only a template for an ideal functionality for E2E voting. This emphasizes the fact that further research will be required to establish a precise formulation of this ideal functionality and 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 E2E ideal functionality has a number of interfaces that are described below. The interfaces 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 this party).

The administrator interface is the interface that is given to the operator of the election system. This interface enables the administrator to setup an election, an action that involves the description of the ballot together with 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 and so on). 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 the interface that 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 to the voter some feedback information that will enable the voter to verify that her vote has been correctly recorded and included in the final tally.

The auditor interface is the interface that is provided to any interested party and it allows the 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, given the feedback of a voter, it will enable a response on whether the vote that has been recorded by the ideal functionality matches the original intent of the voter.

The adversary interface enables the adversary to learn and influence the way the ideal functionality operates in a number of ways. First 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 the voters' submitted votes.

Ideal Functionality

We now describe a general template describing how the functionality reacts when given input in any of its available interfaces. As discussed above, the description we give here should be interpreted as a general guide for expressing the syntax and properties of an ideal functionality for an E2E verifiable e-voting system rather than the definitive final formulation of such functionality. Using the below as a basis a number of different functionalities may be derived that share the same interfaces but differ slightly in the way they operate when given inputs to one or more of their interfaces.

The functionality recognizes a number of 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 that is calculated based on the recorded votes. Intuitively the $P(.,.)$ predicate captures the fact that we may implement an E2E voting 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, y if $x = y$ then it holds that $P(x, y)$. We proceed to the description of the actions taken by the functionality.

- Given an initialization message in the administrator interface it will parse it to extract the list of eligible voters and the description of the ballot. It will forward the initialization message to the adversary. Assuming the adversary enables it, will check that the list of eligible voters is a subset of the list of parties it can identify and it will respond with success to the calling party. Otherwise it will respond with failure.
- Given an open polls message in the administrator interface it will switch its internal state to accepting votes after taking permission from the adversary.
- Given a cast vote message in the voter interface it will parse it to extract a choice for the election's questions. Then, assuming that the administrator is not corrupted, it will notify the adversary about it without communicating the choices of the voter; In case the administrator is corrupted the choices of the voter will be provided to the adversary. When activated, the adversary will decide whether a response back to the voter is to be provided; Given permission, the ideal functionality will check that the voter is among the eligible voters and it will check that the vote is valid given the election's definition. If this is the case it will store a record with the voter's identity and the choices she selected. Then, it will request the feedback for the voter to be specified by the adversary. This feedback will be returned to the voter in order to signify that the ballot-casting submission has been accepted. The adversary is free to provide the feedback string however the functionality will restrict it to be unique: the adversary will not be allowed to use the same feedback string twice. In any other respect, the precise structure of this string is entirely left to the discretion of the ideal world adversary. The feedback string is appended to the voter record and is kept in the local state of the ideal functionality.

- Given a close polls message in the administrator interface the functionality will switch its internal state not to accept votes anymore after taking permission from the adversary. Then, it will calculate the final result based on the records that are kept for the eligible voters that have voted. It will forward the tally to the adversary. This will be called the *calculated tally*. The adversary will respond with a possibly modified tally that will be referred to as the *final tally*. If the administrator is honest the final tally will be forced by the functionality to be equal to the calculated tally (independently of what the adversary suggested). Both calculated tally and final tally will be recorded in the local state of the ideal functionality.
- Given a read tally message in any interface it will forward the request to the adversary. Assuming the adversary enables it, it will return the final tally (note that this may be different from the calculated tally in case 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 will parse it to extract 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 will change the voter's choice in the table where it keeps the voter records making a note that the voter's choice has been switched maliciously.
- Given a verify election record in the auditor interface, the functionality will parse it to extract a voter feedback string. It will forward this message to the adversary. If the adversary enables it, it will attempt to identify the record of a voter that was given such string and then return to the auditor a single bit signifying whether the original voter intent has been tampered with or not. Furthermore, it will utilize the predicate $P(.,.)$ and it will pass to the predicate the calculated tally and the final tally. It will return to the callee the output of the predicate.

Security Characteristics

Using the template ideal functionality of the above section we next argue how a number of 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 when this is given to the functionality in the voter interface provided that 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, 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 and naturally no e-voting implementation can be expected to protect privacy in these scenarios.

E2E Verifiability. The verifiability that is 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 in the case that 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 will ensure the correctness of the tally in a statistical sense. We note that this is one of the distinctions of this ideal functionality compared to ideal functionality for "secure multiparty computation" which is a standard cryptographic notion and the output, if produced, is guaranteed to be correct independently of the actions of the adversary.

Receipt-freeness. Receipt-freeness ensures that the voter does not obtain any feedback from the system that can be used to identify how she voted. Specifically, observe that, provided the administrator is honest, the feedback string is generated by the adversary and is independent of the voters' choices. It follows that this feedback cannot carry any voter choice information. On the other hand, if the administrator is corrupted observe that this property is not preserved: the feedback may be chosen as a function of the voter's choices and hence the receipt-freeness property cannot hold. Note that if a voter is corrupted after her vote is cast, the template ideal functionality will not divulge the voter record that it has kept 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.

6.1.2 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 it still provides some security guarantees even in the case they are corrupted. We examine the effects of corruption from different perspectives.

Corrupt Election Administrator

A corrupt election administrator has a drastic effect in the way the ideal functionality operates. Note that in the template ideal functionality we express by a single entity the administrator of the election. In a protocol implementation that realizes the ideal functionality, the administrator may be implemented by a quorum of parties/trustees that collectively share the responsibility of the election administration. In such 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-freeness is lost and moreover the 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. Note that the functionality enforces that the feedback provided, uniquely identifies each voter and thus any auditor that has at her possession 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 either at the onset of the system operation or even during the ballot casting process. It is expected that an adversary controlling a certain number of voters is also able to shift an equal amount of votes in the final tally. However, the power of such an adversary is restricted to this level of capability 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. In more detail, 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 world.

6.1.3 Absent Security Properties

In the template ideal functionality a number of security aspects are not dealt with explicitly. The fact that 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. Given that 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 as they can be desirable at least to a certain extent; whether a scheme is suitable for deployment in a certain context may depend also on its performance on these additional security characteristics. We elaborate on them below.

Denial of service attacks. The template ideal functionality enables the adversary to prevent voters completing the ballot-casting protocol and may also prevent the tally from becoming available. From a definitional point of view, expressing such level of security is feasible by assuming certain qualities of the underlying communication and message passing mechanisms that are employed in the implementation. One way to extend the functionality to capture such a setting is to oblige the adversary to deliver its messages by certain deadlines; in such case the functionality may go ahead and deliver messages in its output interfaces without requiring the adversary to enable suc messages. In order to do this formally, a notion of time will have to be introduced in the model. This may be achieved by introducing a global clock functionality based on which such 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 mediating in a more strict way 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 template functionality is applicable to the setting where the list of voters is predetermined, assumed to be public and the adversary may 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, note that the functionality can produce the list of identities that have participated in voting and in such case it can be possible for auditors to perform a check in order to verify whether the active voters correspond to real persons (e.g., by selecting a small random sample of them and performing an in-person interview).

Privacy beyond the voters' choices. The template functionality as written leaks to the adversary a number of voter behavior specific aspects including the time that a certain voter casts a ballot as well as the final list of voters that have participated in the election. As mentioned above, this is also a security feature as 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. As written, the template functionality does not provide any mechanism to resolve disputes between sets of voters that claim their voter records have been compromised and an election administrator that claims the particular 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 (e.g., if the number of complaints is below a 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 ideal functionality present some information about the corruption state of parties (currently no such information is divulged to the auditor).

6.2 Contextual Analyses of Primary E2E Protocols

In this section we overview 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 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 maintains unambiguously the transcript of the election. This is the transcript that will be used by the auditors to validate the election result together with feedback information collected from the voters. We remark that the above are in some sense minimal assumptions. If the voters are not able to have a unique view of the election transcript then it is impossible to have verifiability overall and the the template ideal functionality would immediately disqualify a system where an adversary can disrupt the voters' unique view of the election transcript.

6.2.1 Demos

Demos is a system that was recently put forth in [**demos**]. 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 in a public bulletin board. The administrator then distributes ballots that enable voters to cast their vote by essentially pointing to specific encoded information in the public bulletin board. The election terminates by having the administrator produce a proof in the bulletin board 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 at random one of the two (say "A" or "B"). While the A-B decision does not affect the way 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 utilized 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 is only needed to submit 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 in the public bulletin board. In order to attack verifiability effectively the administrator will have to guess the choice made by the voters in the A-B decision. While this may be feasible for just a handful of voters, it is shown in [**demos**] 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 potential has the potential to realize some useful instantiation of the template ideal functionality given the proof arguments presented in [**demos**] assuming there is sufficient entropy in the A-B decision that is performed by the voters.

6.2.2 Helios

Helios is a system that was introduced in [**helios**] and 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 utilizing 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 (be it a PC, smartphone or other system). This is performed at a moment where the device has produced a ciphertext; the voter is asked whether to cast it or audit it. In case of an audit the coins 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). When the voter has performed a number of audits and kept the results she can cast her encrypted ballot. With each ciphertext that is produced a "smart ballot tracker" information is provided as well, that is the hash of the ciphertext. This can be used by the voter to verify that the ciphertext that was generated is the same one that appears in 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 on the bulletin board and finally (iii) verifying the zero-knowledge proofs that ensure that the encrypted ballots are properly encoded and that the ciphertext processing (be it homomorphic or based on mix-nets) 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 [**zeus**].

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

6.2.3 Norwegian System

The Norwegian system [**norwegian**] is based on a protocol developed by Scytl. The administrator will generate for each voter a list of receipt codes, one for each possible choice. These are handed to the voters ahead of time. The voters submit their encrypted vote 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 is argued to ensure the voter that the system has correctly recorded her intent without violating the privacy of the voter. The reason for that is that the code itself is pseudorandomly dependent on the choice of the voter and it is 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 while it is hard 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 thus verify that the system correctly recoded her intention.

Contrary to Helios or Demos, verifiability in this system can only hold provided that the adversary does not control the servers and the device used by the voter to cast the ballot. Indeed, in such case it is possible that the system deceives the voter that the proper choice was recorded while something else has been recorded instead. It follows that the verifiability guarantee for the Norwegian system is weaker (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, such restrictions would be preferable to be avoided if possible).

6.2.4 Remotegrity

Remotegrity [**remotegrity**] is a front-end system for Internet voting that can be combined with a back-end (such as Scantegrity-II [**scantegrity2**]) to offer a complete e-voting system with end-to-end verifiability guarantees.

During initialization the system administrator precomputes a number of encoded ballots as well as commits to a properly formed set of tables of commitments in a public bulletin board. For each candidate choice there is a unique vote-code that is assigned to each voter and 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 will be revealed at the end of the process. Cast ballots on the other hand record a certain vote-code and the corresponding election choice will be revealed in a way that the correspondence to the voter will remain 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. This is tight, as it is easy to see that in case the adversary completely controls the randomness used in the challenge stage she can produce any tally that is independent of the voter's intent.

Regarding the randomness used there are several proposals for natural or human public phenomena that have been proposed for use in this context (for instance, stockmarket end of day quotes, weather measurements are potential candidates). We stress that using such sources of randomness is not straightforward as the randomness quality is limited and a proper extraction of uniform randomness needs to be applied.

6.2.5 RIES

The Rijnland Internet Election System (RIES) was used between 2004-2006 for elections in Dutch District Water Boards and in 2006 for the Dutch expats in the parliamentary elections. The system 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 vote-code) and maintains that vote-code as a receipt for the voter. Meanwhile the administrator decrypts all votes and posts them to the bulletin board in some predetermined order (that also depends on the symmetric encryption key).

Note that it is possible to calculate the final tally using the information that is put in the bulletin board by matching them with the hashed vote-codes. However, the system is not a candidate for realizing a reasonable variant of the template ideal functionality given that it produces receipts that unequivocally identify the way that voters choose their selection during ballot-casting. Indeed, the system is not receipt-free and hence cannot be used as a reasonable implementation of the ideal functionality.

6.2.6 Wombat

The wombat system [**wombat**] falls in the same category as Helios and Zeus with the exception that it uses an onsite ballot-casting setup that also preserves the original voter intent in paper. The voters generate ciphertexts with the aid of a device with a printer that presents the voter choice both in ciphertext form as well as in plaintext form. The voter may choose to audit the combination using the same exact logic as the audit or cast mechanism described above in the Helios section. 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 published in a bulletin board.

The voter retains the encoding of the ciphertext that was cast as a fingerprint to match it with the information in the bulletin board (this plays the same role as the smart ballot tracker in the case of Helios).

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 themselves. Overall the same verifiability and privacy arguments made in the case of Helios would apply here (while taking into account the fact that wombat is an on-site system).

6.2.7 vVote/Prêt à Voter

The vVote system is an adaptation of the Prêt à Votersystem for the Australian state of Victoria elections [**vvote**]. It retains the general structure of Prêt à Voter[**pret-a-voter**] while it modifies the way the encryption of voter choices is performed so that it suits the style of elections of Victoria.

Recall that Prêt à Voterpaper 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 in the right-hand-side while the names of the candidates appear (randomly permuted) in the left-hand-side. After marking, the left-hand-side is destroyed while the right-hand-side is scanned so that it is posted in the bulletin board as well as it is kept as a receipt. The right-hand-side contains also the encoding of a ciphertext that enciphers the permutation that was used in the ballot.

Verifiability can be based on an audit-or-cast process on the printed ballot and the verification of the process in 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 in the the ballot. The voter can also track her ballot in the bulletin board (recall that the right hand side contains an encoding of the ciphertext). Finally the result is revealed after a mix-net operation which includes zero-knowledge proofs of knowledge 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 Voteris similar to that of Helios and Wombat.

6.3 Realizing Ideal Functionality

In this section we describe some general directions and tools regarding the realization of the ideal functionality. We start with the description of relevant cryptographic tools and then we move on to describe how it is possible to mechanize the security proof of realizing the ideal functionality. A case study for one system is presented. Finally the case for open protocols and software independence is made.

6.3.1 Commonly Used Cryptographic Tools

A number of cryptographic tools play an important role in the design of e-voting schemes. We provide a list of some of these tools that have found use in the e-voting design of the systems we overviewed in the previous section.

Commitment schemes. A (non-interactive) commitment scheme is comprised of two algorithms $(\mathrm{Commit}, \mathrm{Verify})$ and enables a party to produce a pair $(\psi, \rho) \leftarrow \mathrm{Commit}(m)$ 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 interested party can verify that indeed the process was done correctly by running the predicate $\mathrm{Verify}(\rho, m, \psi)$ and accepting the opening provided that the predicate returns true. It is possible that the two algorithms are parameterized by a public parameter p that is available to all parties. In this case an algorithm may be added in the description that generates the parameter p. 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 open it to a different value m'.

Public-key encryption. A public-key encryption is comprised of three algorithms $(\mathrm{Gen}, \mathrm{Enc}, \mathrm{Dec})$. The algorithm Gen produces a pair of public and secret-key denoted by (pk, sk) while $\mathrm{Enc}(pk, \cdot)$ on input plaintext m it samples a ciphertext ψ that corresponds to m. Finally, the algorithm $\mathrm{Dec}(sk, \cdot)$ returns the plaintext that corresponds to the input ψ. The security model requires at minimum that an adversary that is given the public-key pk, is incapable of distinguishing between two ciphertexts that are sampled corresponding 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 $\mathrm{Enc}(pk, \cdot)$ algorithm. First the space of plaintexts is endowed with a group structure over a binary addition operation $+$. Then, given $\psi_1 = \mathrm{Enc}(pk, m_1)$ and $\psi_2 = \mathrm{Enc}(pk, m_2)$ it is possible to compute a ciphertext ψ that belongs to the encryptions $\mathrm{Enc}(pk, m_1 + m_2)$. Furthermore, we also require that if at least one of ψ_1, ψ_2 is sampled uniformly over all ciphertexts then the output ψ is following 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 T can be used to derive the tally of an election in case each plaintext v_i corresponds to an election choice made by the i-th voter. (ii) Given $\psi = \mathrm{Enc}(pk, m)$ it is possible to "refresh" the randomness of ψ by processing ψ together with $\mathrm{Enc}(pk, 0)$. The resulting ciphertext is uniformly distributed over the ciphertexts encoding m.

Secret-sharing. A secret-sharing scheme is comprised of two algorithms $(\mathrm{Gen}, \mathrm{Rec})$ 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 that is provided by the Gen algorithm.

Threshold encryption. A threshold (public-key) encryption is equipped with a multiparty protocol that implements the $\mathrm{Gen}()$ public/secret-key generation algorithm between 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 $\mathrm{Dec}()$ by the trustees so that each one of them 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 it may be participating on 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, 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 is 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 simply 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 $\mathrm{Shuffle}()$ 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 $\mathrm{Dec}(\psi_i)\rangle$ and similarly M' the sequence of plaintexts that in the i-position is equal to $\mathrm{Dec}(\psi'_i)\rangle$. It should hold that there is a permutation μ over n elements such that the j-th element of M equals to the $\mu(j)$ element of M' for all $j = 1, \ldots, n$.

Given the $\mathrm{Shuffle}()$ algorithm observe that we can apply it sequentially on the same sequence of ciphertexts. If a sequence of m servers apply $\mathrm{Shuffle}()$ 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.

6.3.2 Other Potentially Useful Cryptographic Tools

There are several other cryptographic tools that have not yet seen much application in the context of E2E protocol design. Several are briefly summarized here, in part because they represent opportunities for novel research directions in E2E 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 hash chain is a data store that holds a ledger of sequential records in a cryptographically secure fashion [**HC**]. 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 by which to record privacy-preserving election logs for post-election audit, construct digital ballot boxes, craft public bulletin boards that contain evidence of an election's correctness and security properties, and more [**HC-for-elections**].

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 a number of 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 [**NLHC**]. They are useful in the context of disconnected devices with causally or temporally connected ledgers (e.g., disconnected DRMs used in a given election) [**NLHC-for-elections**].

A blockchain is a distributed data store that holds a ledger of transactions in a cryptographically secure fashion [**blockchains**]. It is a kind of distributed hashchain, wherein multiple computers compute and communicate to determine what the next data element of the hashchain/tree using a consensus protocol. Blockchains often rely upon a cryptographic work factor to determine consensus as well as to prevent manipulation of the ledger by powerful adversaries.

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

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 parties gaining any knowledge about the inputs provided by the parties, expect what can be determined from the output of the computation [**MPC**].

Since systems compute on encrypted data, computation can take place on public systems, such as in a public cloud. Moreover, since systems can communicate using a published public protocol to collaboratively compute, parties 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 [**LSS**]. Certain operations, such as addition or logical-XOR can be performed locally, but operations such as multiplication or logical-AND require a network communication between the parties. Consequently, the computational overhead of MPC is large, and the cost is measured in orders of magnitude slowdown with respect to computing in the clear.

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

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 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 its encrypted data [**CryptDB**]. Consequently, FE may prove useful in the storing of, or computing on, election data like ballots, audit logs, and more.

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 protocols [**IBE**, **ABE**].

Fully homomorphic encryption. Fully homomorphic encryption (FHE) is a more powerful version of the aforementioned partial homomorphic encryption (in the form of additive or multiplicative homomorphic encryption) cryptosystem. FHE permits arbitrary computation on encrypted data, but is prohibitively slow (several orders of magnitude slower than plaintext computation) and ciphertexts are enormous (dozens to hundreds of megabytes in size).

It is unclear if FHE has potential application in the context of E2E protocols, but given the swelling interest in FHE and the availability of open source implementations of FHE libraries, surly we will see its applications, particularly in the context of cloud deployments of election systems.

Verifiable computing. Verifiable computing is a new area of R&D which focuses on the offloading of computation to an untrusted system (say, an untrusted DRM evoting machine or a cloud hosting service), and yet 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 [**TPM-stuff**]. These verifications are either interactive, which require the client to interact with the worker to verify the correctness proof, or are non-interactive protocols which 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 kind of data volumes mandated by public elections. It is to be expected that new results in both verifiable computing, and E2E protocols that presume the existence of efficient verifiable computing, will be forthcoming.

6.4 Formal Mechanization of Ideal Functionality

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

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

There are several other excellent useful environments that should be considered for future mechanization work, such as higher-order frameworks such as PVS, Isabelle, and HOL, and protocol-centric tools such as F, CVK, and ProVerif, 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 there are several libraries 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, which achieve specific functionalities through intricate combinations of several primitives. These developments have expanded significantly 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 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. Cryptol version 2 is open source and available under a 3-clause BSD license. Galois's goal is that it becomes a widely adopted standard for expressing cryptographic algorithms.

Our recommendation is to experiment with each of these alternatives, formalizing ideal functionality and at least one E2E protocol. Based upon that case study, an objective decision can be made about which toolset to use in the long run.

6.4.1 Recommendations

Based upon our experience, the most promising approach is 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, and their implementations' (in LLVM or JVM), correctness, use EasyCrypt to reason about algorithms' and protocols' security properties, and use F to specify and reason about protocols that need a Javascript or .Net implementations.

6.5 Specification of Open Protocols

E2E 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) is 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 a straightforward proposition. Verifing the protocol's 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 an implementation of a protocol 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 [**nqsb-TLS**] in a pure functional style exemplifies the kind of reference implementation that should be written for an E2E protocol.

6.6 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. A 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 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 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 elected 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. See [**connecticut-audit**] for more details regarding the procedure that is followed.

Chapter 7

Architecture (Joe K./Dan) (100%)

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.

7.1 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.

7.1.1 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.

7.1.2 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.

7.1.3 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.

7.1.4 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 [33], 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* [74], effectively single-purpose application/operating system combinations that run directly atop hypervisors such as Xen [126]. Unikernels, written using technologies such as Mirage [79] and HalVM [114], 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 number of 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 a number of possible threats; we describe the distribution of trust in more detail in the following section.

7.1.5 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 [80] 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 7.1.4, 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 [69], 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 mainstream 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 [67], several JavaScript tools from the Center for Advanced Software Analysis [26], and Microsoft Research's Crypto Verification Kit [39]. 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 [12] 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 [92], 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 [70] 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.

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

[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.

[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.

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" [115], 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" [125].

7.1.6 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.

7.1.7 Availability

E2E-VIV systems must exhibit high availability; Chapter 5 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 a number of 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.

7.1.8 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.

7.2 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 7.1 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 [19]. 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 generate correct implementations from specifications. Second, the implementation can be developed "by hand" with a set of

```
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 7.1: A specification of the possible variants for an E2E-VIV system.

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 7.1.5, 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 7.1.5, 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) [132] 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).

7.3 Primary Architectural Variants

Given the many dimensions of the architectural feature model, and number of choices in each, the number of possible architectural variants for the system is large. 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).

7.3.1 Mirrored Servers

One possible architecture, which features centralized authority as its primary defining characteristic, is the "mirrored servers" architecture depicted in Figure 7.2. 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.

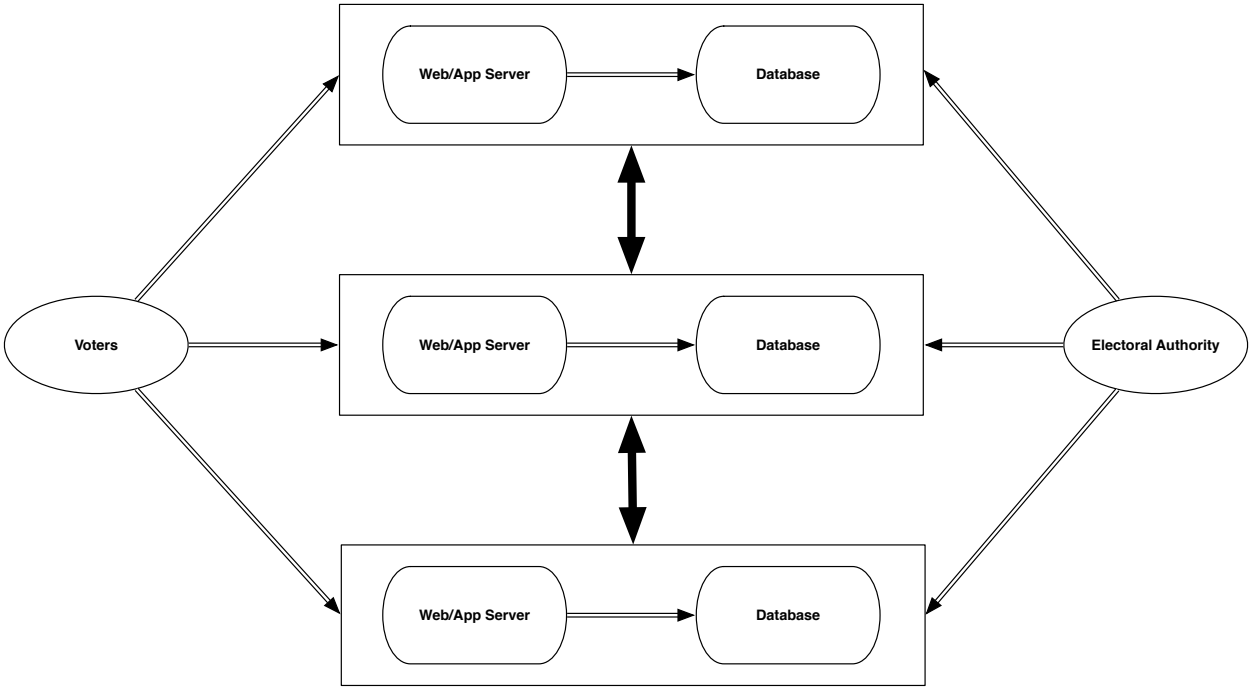

Figure 7.2: An architecture with centralized authority and mirrored servers.

7.3.2 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 7.3, 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 7.3—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 7.4) 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.

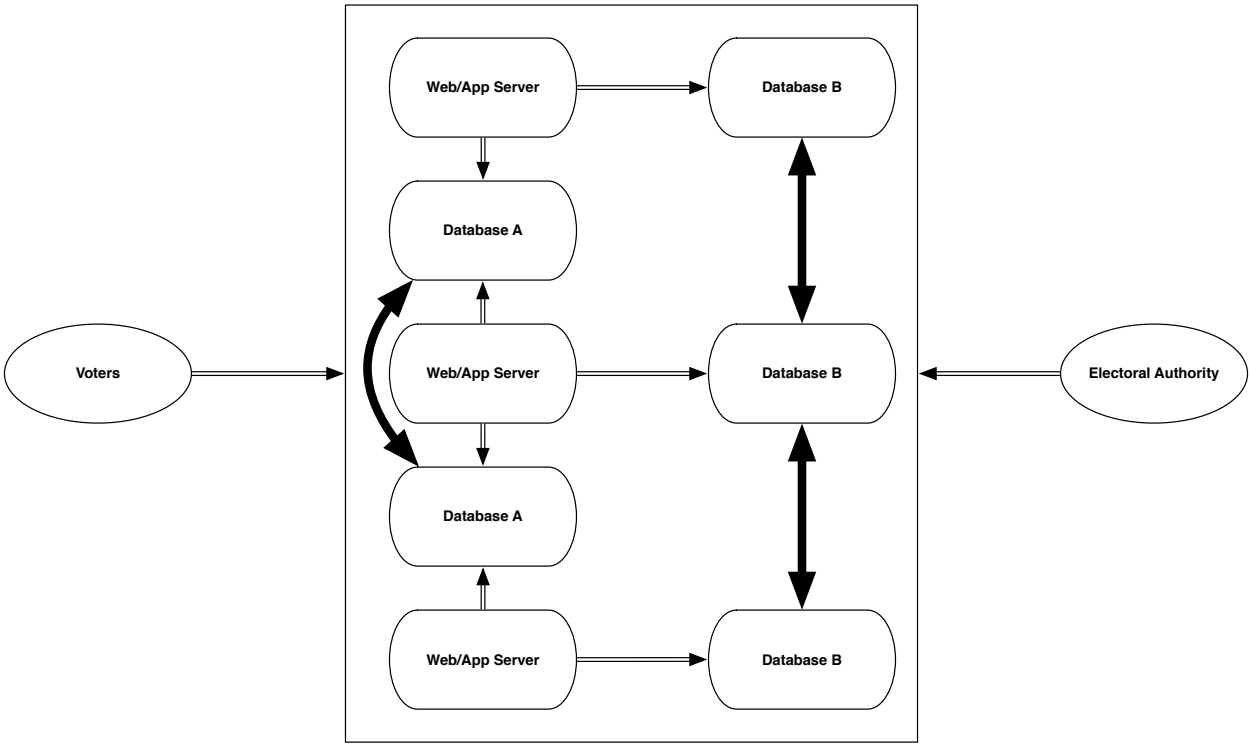

Figure 7.3: An architecture with a large fixed set of servers.

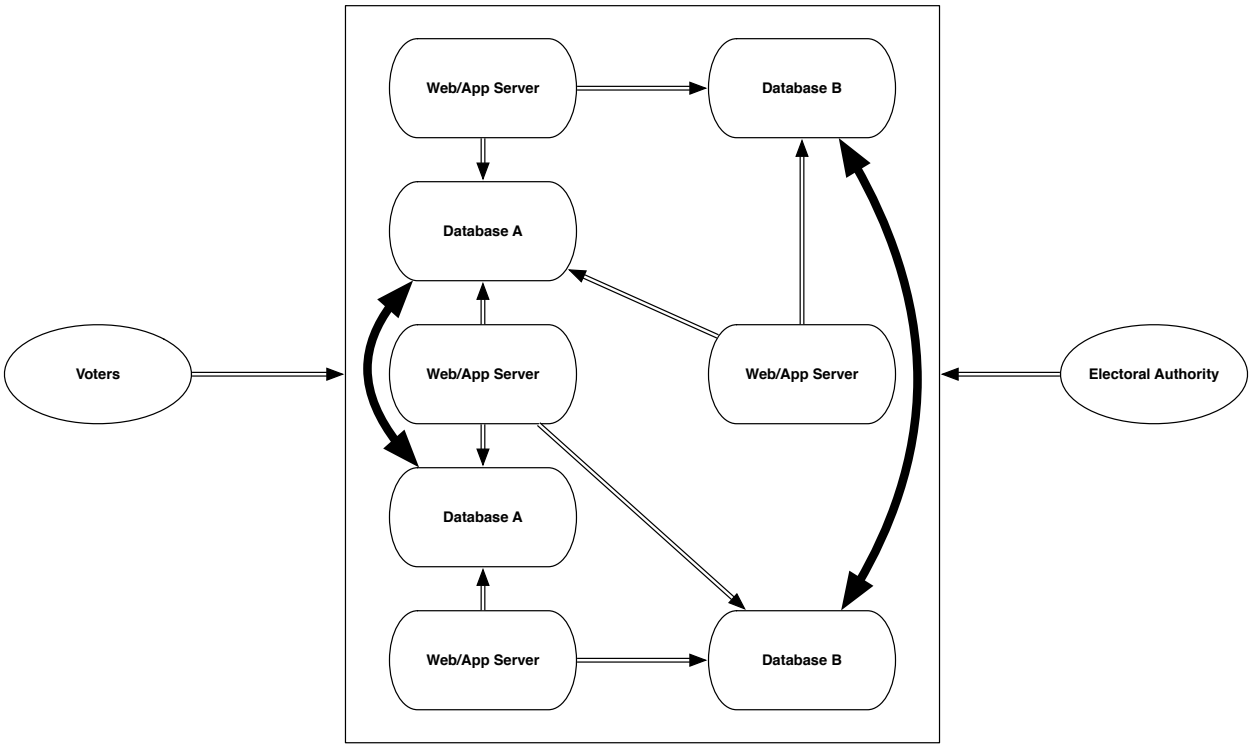

Figure 7.4: The same fixed set of servers as in Figure 7.3 performing a different allocation of tasks.

7.3.3 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 7.5 and 7.6 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 7.5, 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.

7.3.4 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 7.7 depicts a peer-to-peer system with a number of 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.

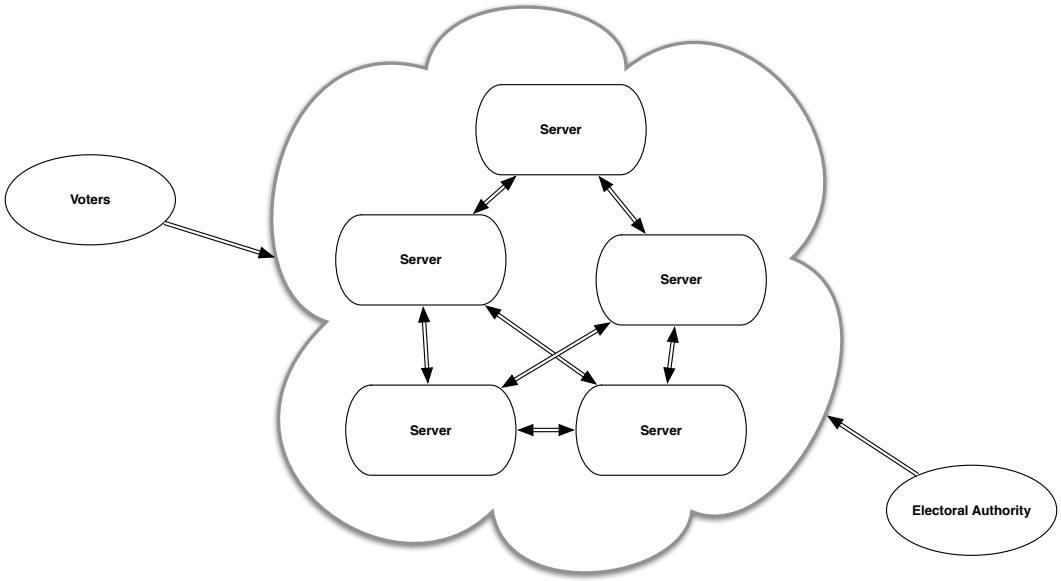

Figure 7.5: A dynamic cloud architecture with a small number of servers.

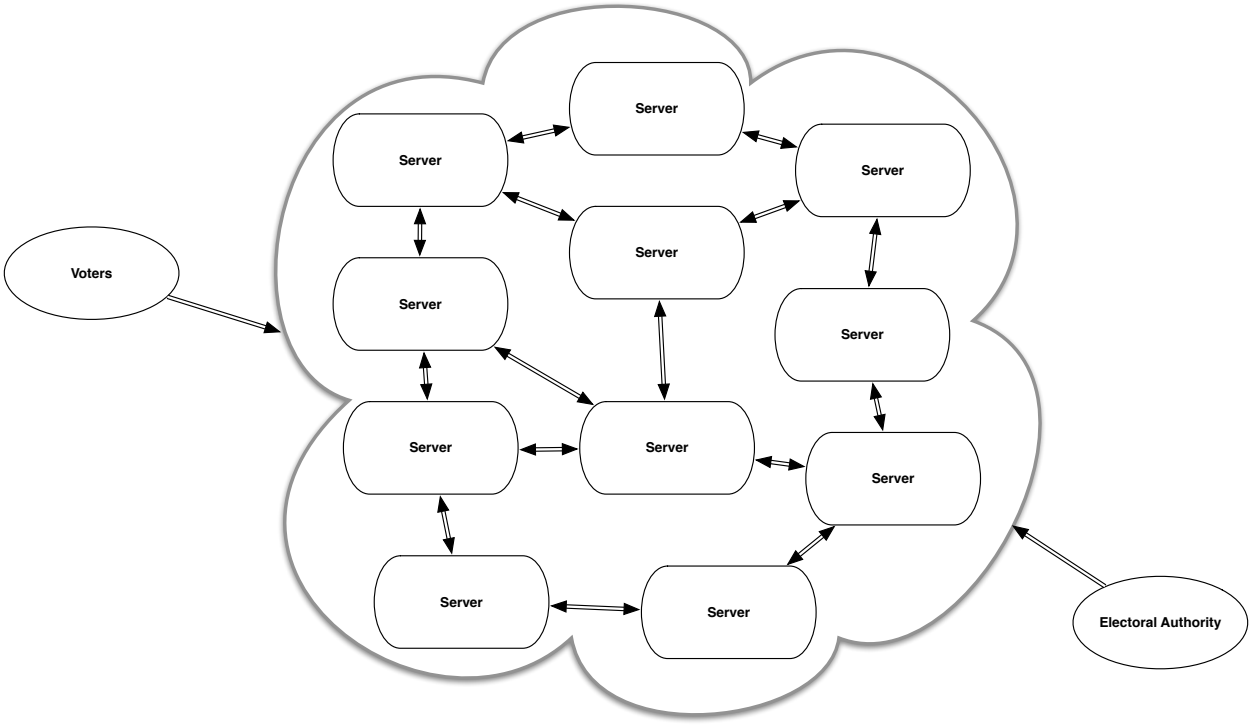

Figure 7.6: A dynamic cloud architecture with a larger number of servers.

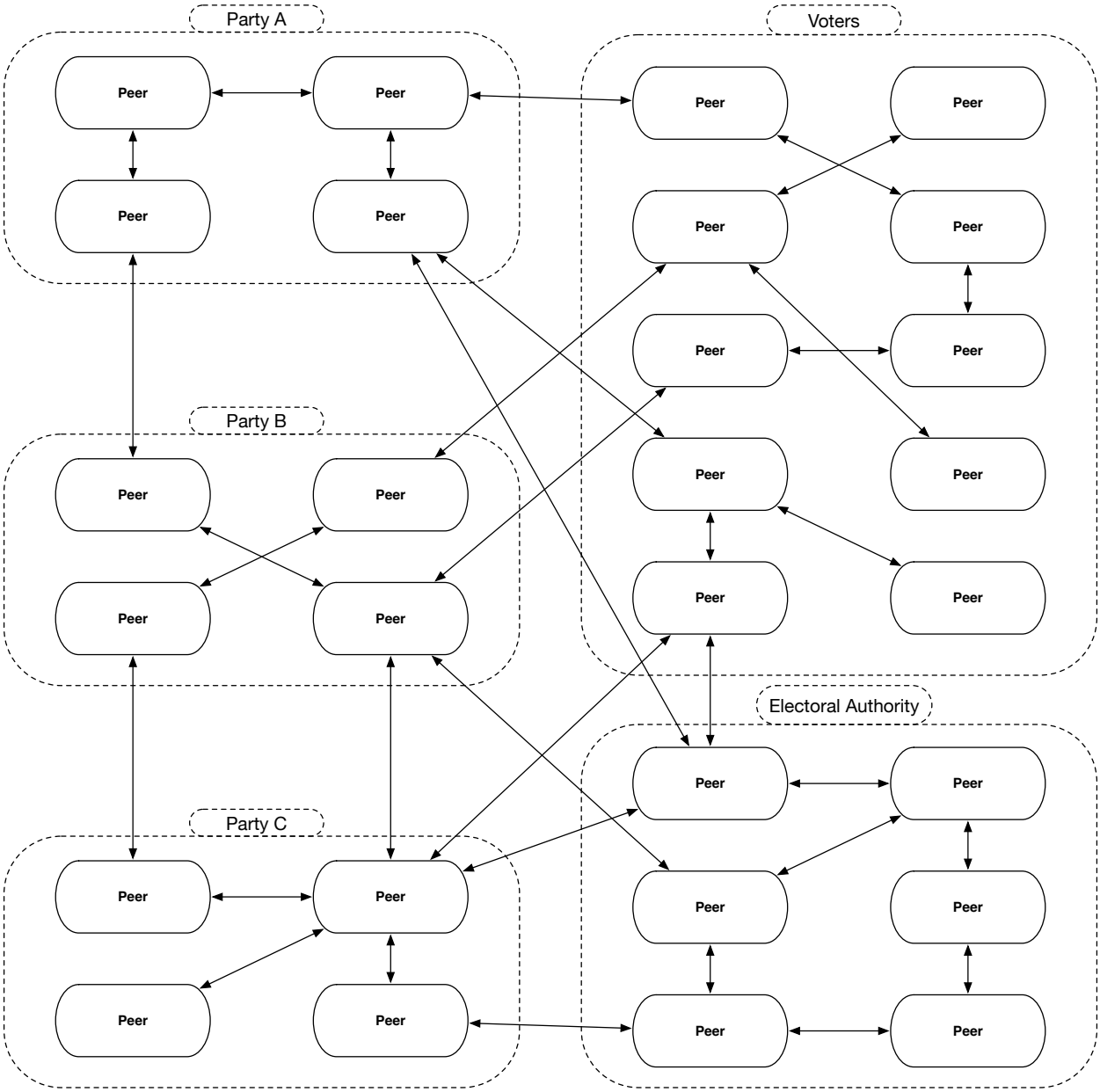

Figure 7.7: A peer-to-peer architecture.

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.

7.4 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 could be designed and implemented. It is not clear which of the primary variants would be the "best" option, nor is it clear exactly what criteria would be used to make that determination among multiple architectures that fulfill all the E2E-VIV requirements. Further research and experimentation is therefore necessary to determine a suitable path forward for E2E-VIV implementation and deployment.

Chapter 8

Rigorous Software Engineering (Joe K./Dan/Adam) (40%)

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.

Fix that last sentence depending on what the end of the chapter actually ends up look-

8.1 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.

8.1.1 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 (see **??**) 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.

8.1.2 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 5, and the informal requirements document itself is in Appendix A.

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.

8.1.3 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.

8.2 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.

8.2.1 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 7.3.

Some research has been done with respect to verifying software implementations or hardware simulations against architectural specifications [2, 72] 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.

8.2.2 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 [78]: 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.

8.2.3 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) [42] 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 a number of 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.

8.2.4 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.

8.3 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.

8.3.1 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 [65], however more sophisticated coverage measures such as branch coverage are more promising [56].

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 [111].

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 [127], 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) [41], and symbolic execution with constraint solving (calculating new inputs to avoid repeating execution paths that have already been tested) [58]. 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 [128].

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.

8.3.2 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 [27].

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 [14, 90].

8.3.3 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.

8.3.4 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.

8.3.5 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 [47]. 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 [117] and have other positive effects such as higher developer job satisfaction and improved team dynamics [34].

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.

8.3.6 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 [14], 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.

8.3.7 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) [124] 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 [51]. 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.

8.3.8 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.

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, it is a good practice to have CI 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.

8.4 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.

8.4.1 Domain Modeling

For domain modeling, we recommend the Business Object Notation (BON) [124] 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" [82]; 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 in Appendix A 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) [89], Z [36], the B Method [76], and the Rigorous Approach to Industrial Software Engineering (RAISE) [96] are all well-established domain analysis methods with textual representations, well-defined semantics, and tool support.

8.4.2 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) [48] or the OMG Systems Modeling Language (SysML) [84], 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 [87] for AADL, also support automated verification.

Programming languages for which there is an obvious best choice of source code specification language include Java (JML [71]), C# (Code Contracts [35]), and C (ACSL [5]). Some implementation languages, such as SPARK [107] (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 [8], Coq [112], Event-B [3], VDM [89], and Z [36], 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 [11], typed π-calculus as used by ProVerif [19] and CryptoVerif [38], Casper [73], EasyCrypt [43], and Scyther Protocol Description Language (SPDL) [37]. 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.

8.4.3 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 [23] or the C dialect supported by the Verified Software Toolchain [122]), C# (excluding unsafe code), Eiffel, Erlang, Gallina (the executable specification language for the Coq proof assistant [112]), 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.

8.4.4 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 [32] for Java and StyleCop [109] 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 [49] and PMD [91] for Java, ReSharper [98] for C#, Eiffel Inspector [44], and SPARK Pro [108].

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 [86] for Java with JML specifications, Frama-C [52] 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) [101]; this type of analysis is particularly useful when verifying the correctness of cryptographic algorithms, which are pervasive in E2E-VIV systems.

8.4.5 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 [113], 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 [45] for Java, OpenCover [85] for C#, and GNATcoverage [57] 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.

8.4.6 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 [8], PVS [103], Spin [123], and UPPAAL [120]. 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 a number of 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.

8.4.7 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 [54] for Git, BitBucket [18] for Mercurial, and SourceForge [106] 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.

8.4.8 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 [66], JetBrains's YouTrack [129], and Fog Creek Software's FogBugz [50] 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 [118] (along with its spinoff project Apache BloodHound [10]) and Redmine [97] 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.

8.4.9 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 [110] 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 [68] and TestNG [16] for Java, NUnit [81] for C#, and HUnit-Plus [64] 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 [95] 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 [9], 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 [131] for Java/JML, AutoTest [77] for Eiffel, and PEX [116] for C#.

8.4.10 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) [60]. 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.

8.5 Evidence-based Elections Technology

11: Reflect upon the fact that no voting systems in existence today use even the basics that we have covered. Reflect on why this is the case: lack of capability in existing vendors, no pressure from NIST/EAC to do better, etc. Reflect upon how different this is in safety-critical domains like at JML and Airbus.

8.5.1 Measuring and Assessing Quality

8.5.2 Interpreting Evidence for the Non-expert

Chapter 9

Feasibility (*Joe, David, et al.*) (25%)

In the first fifty-odd pages of this report, in Chapters 2 through 5, we laid out the motivation for, history of, and requirements on a remote voting system that both experts demand and the public will trust.

Then in Chapters 6 through 8 we described the necessary cryptographic, architecture, and engineering foundations, tools, and techniques necessary for designing and building a system that fulfills the criterion set out in Chapter 5's requirements.

But just because it seems *possible* to design and develop such a system, does not mean that it is *feasible* to do so.

This chapter analyses the question of feasibility from several dimensions, 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 semi-orthogonal dimensions, we summarize with an integrated feasibility analysis, focusing on the question of "Does it makes sense to practically tackle the problem of E2E-VIV at this time?"

In what follows feasibility is determined by a principled examination of the current state-of-affairs based upon the peer-reviewed literature, extensive conversations with those responsible for elections, multi-year dialogs with election verification activists, and decades of experience in the design and development of secure high-assurance systems.

The final determination of feasibility is not up to us, as authors of this report. This is a decision, for the most part, that must be made by those organizations with resources that can be brought to bear on this problem. The aligners with that decision—primarily the activist community and legislatures—while important, are not critical at this point in time, given the deployment velocity of unverifiable internet voting systems worldwide.

9.1 Technical Feasibility Analysis

12: If E2E-VIV is not technically feasible, then we are DOA.

13: Briefly summarize the main outstanding research and engineering challenges, as those are what a phase 2 of this project focuses upon.

9.1.1 Correctness

9.1.2 Security

9.1.3 Usability

In order to effect these goals, a demonstration system that mimics a voter's interaction with an E2E-VIV system has been developed by Galois. That system is a variant of the STAR-Vote system designed by Wallach et al. [15]. STAR stands for Secure, Transparent, Auditable, and Reliable. STAR-Vote is an end-to-end verifiable ballot marking device. As such, it is not designed for, or meant to be used for, Internet voting. But insofar as its voting process is identical to that of most of E2E-VIV election schemes in the literature, we decided to use it as a demonstration vehicle for usability and accessibility experiments.

15: Recall that this includes accessibility. Reflect upon the split personality of the Demos protocol for usability for the typical set of voters.

The Galois STAR-Vote implementation has a web-based UI, thus can be used and demonstrated remotely for interactive and non-interactive experiments to gather both qualitative and quantitative feedback. Several variants of STAR-Vote have been implemented for UX testing. These variants include simple changes—like different typeface choices and sizes, background colors, supporting images, help text, mouse pointer graphics, etc.—as well as more complex changes—like different voter, challenge, and audit workflows.

In an interactive, qualitative experiment, a facilitator and a voter communicate using a video chat system such as Skype and the voter shares their desktop with the facilitator. Optimally, the facilitator is someone who is deeply familiar with the issues of E2E-VIV systems, is familiar with STAR-Vote, and has expertise in usability and accessibility. The voter then uses (one of several variants of) STAR-Vote, voicing their thoughts and feelings about their experience in real-time. After the voter has completed their participation in the demonstration election, the facilitator uses a script to query them about their impressions.

For a non-interactive, quantitative experiment, voters will be solicited via social media, mailing lists, etc. to experiment with (variants of) STAR-Vote. Sample voters in these experiments are given ample information about what kinds of information is being collected about their behavior so that 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 their web browsers and on the STAR-Vote server. Most of this data is akin 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 and 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?

9.1.4 Availability

16: Reflect upon the current state-of-the-art in providing availability for core services on the internet. How expensive and difficult is such a deployment? Do the more radical architectures described earlier provide serious alternatives?

9.1.5 Operational

9.2 Non-Technical Feasibility Analysis

9.2.1 Law

17: What are the feasibility

9.2.2 Politics

9.2.3 Fiscal

9.2.4 Research

9.2.5 Development

9.2.6 Operational

9.2.7 Business

9.3 Integrated Feasibility Analysis

20: In the main, politicians want internet voting come hell or highwater. How does phase 2 and 3 look given that vendors are selling product and that politicians do not care about nuances?

21: Reflect upon the cost of previous experiements in developing and trialing internet voting systems. What is the current static state-of-affairs wrt election budgets at the local, state, and national level. There is little more HAVA money, jurisdictions are having to make-do with what they have, and there is little appetite for purchasing new equipment from the existing vendors that they dislike. They really want an inexpensive outsourced product that is secure and usable.

22: What

Chapter 10

Conclusion (Joe K./Susan) (100%)

Internet voting (IV) is a like an impending wave moving towards voters and elections officials. Ensuring that, when it hits us, we have a firm foundation for trust in our elections—particularly in regards to their correctness, security, usability, and accessibility—is paramount.

Given the in-depth analysis summarized in this report, the E2E-VIV Project team has produced the following results, made a concrete set of recommendations, and highlighted next steps for various related communities.

10.1 Results

As promised in the proposal that created this project, several deliverables have been produced.

Primary among them is the "whole product solution" specification contained in Chapter 5. This specification, which comes in the form as a set of mandatory requirements, is useful to several audiences, including:

- *legislators* and their staffs who are interested in crafting laws that relate to remote elections, particularly IV elections and elections for overseas, military, and disabled voters;
- *election officials* specifying, evaluating, or purchasing IV products or services;
- *activists* interested in obtaining a better understanding of, and advocating for, E2E-VIV systems;
- *testing organizations* interested in certifying IV systems;
- *standards bodies* interested in standardizing what constitutes various classes of IV technologies and how rigorous certification of IV systems should be specified;
- *technologists* interested in implementing E2E-VIV systems.

Moreover, this report also includes several other artifacts useful to a subset of these audiences, including:

- a cryptographic foundation with which to evaluate and compare various E2E protocols and E2E-VIV systems (Chapter 6),
- an analysis of the architecture space of E2E-VIV systems (Chapter 7),
- precise recommendations on the state-of-the-art for rigorous engineering of E2E-V systems (Chapter 8),
- a framing for an ongoing discussion about the feasibility of designing, constructing, certifying, legalizing, and operationalizing E2E-VIV systems (Chapter 9), and
- an analysis of the outstanding issues that must be tackled in future stages of this field, including political, legal, research, engineering, and business challenges (Section 10.3).

10.2 Recommendations

Synthesizing all of the above information, the E2E-VIV Project team has come to the following conclusion and set of recommendations, with the caveat that the experts *do not* assert that Internet voting (IV) *must* be pursued, nor do they assert that IV must *never* be pursued. There is clearly no consensus for either of these positions.

Recommendation E2E-V. **Public elections should not be conducted over the Internet using systems that are not end-to-end verifiable.**

Some may say that public elections should not be conducted over the Internet at all, but we have uniform support as a community of the assertion above, as opposed to the kind of "naked", or vulnerable due their defenselessness, IV that we are beginning to see today. Non-end-to-end verifiable systems are irresponsible and should never be used. We all know of the numerous vulnerabilities of Internet applications, and the idea of voting with an unverifiable Internet application is alarming, to say the least.

Recommendation SUPERVISED-FIRST. **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.**

Gaining experience with E2E-verifiability in the simpler in-person setting before attempting the more complex Internet setting seems like a natural and prudent step, but there is another reason for starting with in-person systems. Few jurisdictions are willing to release separate tallies for their in-person and remote voters, but a verifiable system needs to publish a tally to be verified. If only the remote voters use an E2E-verifiable system, then the sub-tally of the remote voters must be reported.[1]

The pragmatism in this recommendation is simpler than the technical justification: The E2E-VIV Project team firmly believe that if we were to assert that Internet voting cannot be conducted with adequate security and assurance, then we will be ignored and see widespread use of deficient, unverifiable Internet voting systems within ten years. Vendors will claim to have solved the security problems, and eager officials will buy these systems. Elections may be altered with no public awareness, and vendors will assert that they were right and we were wrong. If election officials manage to find evidence left by a careless attacker after altering an election, it will be a Pyrrhic victory.

In making these recommendations, we realize that we have created a narrow pathway to adoption and assert that no attempt at Internet voting should deviate from this path. This path is a difficult one to follow. After all, building an in-person E2E-verifiable voting system is no small task. After several years, there will be few, if any, E2E-VIV systems in common use. But any E2E-VIV system is enormously better than the vulnerable Internet voting alternatives.

In addition, we recognize that we get the side-benefit of the deployment of E2E-verifiable in-person systems, which has the bonus of improving the integrity of our in-person voting systems.

Recommendation HIGH-ASSURANCE: **E2E-VIV 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.**

It is sometimes argued that a *strongly software independent* voting system need not be implemented in high-quality software. While theoretically true, this attitude has no practical value.

The need for quality, even for software independent systems, is due to several implications of developing and using a low-quality implementation of any E2E-V voting system:

1. **Privacy violations.** While E2E-V systems can mitigate issues of election integrity well, they do little to mitigate matters relating to privacy. A poorly implemented E2E-V system will be able to identify when something goes "wrong" with the election (e.g., a bug or a hacker causes a vote to disappear), but that is of little remedy if voter privacy has been violated.

[1] It may be possible to mix a small number of not individually-verified remote votes in with fully-verified in-person votes in such a way as to produce a single universally-verifiable tally, but it is difficult to imagine how the reverse might be accomplished in a meaningful way.

27: This is being dropped into this document in this form not because it has been decided, but simply to expedite editing and layout as we push to completion. I am using an edited version of Josh's proposed text and ideas here—they do not reflect Galois's position in these matters.

2. **The impact of programming errors.** It is vastly more likely that a poorly-implemented E2E-V system will have software engineering flaws in design, functionality, security or other areas which trigger failures in verification. These will thereby increasing the burden on the election administrator to mitigate partial failures and potentially have significant impact on the voters' trust in the election, its administrators, apparatus, and outcome.

3. **Security mandates quality.** A low-quality implementation is vastly more difficult to secure in the presence of insider and outsider attack. It may even be impossible to secure it. Security is not a band-aid to tape onto a poorly built system - it is achieved through a combination of rigorous process, method, design, implementation, validation, verification, deployment, and operation.

In summary, the only way to reasonable implement an E2E-V system that is correct, secure, and does not have enormous post-deployment fiscal and trust implications on election officials is to do so using high-assurance software engineering tools and techniques.

Recommendation UNIVERSAL-DESIGN: **E2E-VIV systems must be usable and accessible to the typical set of abled and disabled voters.**

Attempting to design and build a voting system that is usable and accessible to every voter that exists is an imprudent task. The one-in-one-hundred thousand voter who is seriously sight, mobility, and mentally disabled would be assisted at the expense of denying a useful system to 99.9% of the population.

Instead, in addition to fully able-bodied and mentally fit voters, our target audience should include voters that have challenges in vision, hearing, comprehension, and motion yet still can use some kind of computing device. By tackling the low-hanging fruit of universal design - via a qualitative and quantitative testing-based experimentation platform to assess usability and accessibility and following best practices [**materials-at-elections.itif.org**], recommendations [**WAI**, **Section508**, **WAVE**], and standards [**standards**] in accessible UI design and implementation - we will be able to service nearly every overseas and miltiary voter, but also in the long term, the upwards of 84M disabled voters in the U.S.A. [**Brennen**, **CensusData**].

We must also look to, and learn, from the AnywhereBallot and EZ Ballot experiments [**AnywhereBallot**, **EZBallot**] of the AVTI [**AVTI**]. We must engage with the researchers and attendees at CSUN's annual Annual International Technology and Persons with Disabilities Conference [**CSUN**]. Only by having direct engagement with our stakeholders - voters, both abled and disabled - and witnessing their joys and struggles of participating in democracy, can we have any hope of understanding how to develop usable design in E2E-VIV.

Recommendation MOVE-FORWARD: **A recognized dedicated group of qualified subject-matter specialists with credentials in the appropriate subject areas (e.g., verifiable elections, cryptography, cyber-security, high-assurance systems engineering, and the universal design of election systems) should be identified and tasked to design a customized open source E2E-VIV system that fulfills the requirements included in this report and that design should be implemented in a set of high-assurance prototypes.**

Moreover, those prototypes should be objectively evaluated using appropriate validation and verification tools and techniques against both functional properties (correctness and security) and non-functional properties (accessibility, capacity, disaster recovery, fault tolerance, maintainability, performance, reliability, resilience, scalability, and usability) mentioned herein.

Only with the public availability and peer-review of such a focused E2E-VIV system for the U.S. public can we see a path forward toward the use of Internet voting for public elections.

10.3 Next Steps

To fulfill these recommendations, there are several open challenges and next steps for legislators, researchers, engineers, and businesses.

10.3.1 Political/Legal Challenges

The greatest fear voiced by election verification scientists, activists, and E2E-V researchers is that legislators will aggressively mandate the experimentation with—or use of—Internet voting before a correct, secure, open, usable, accessible E2E-VIV system exists. Such a mandate will facilitate only existing vendors whose systems open the door to wholesale election manipulation or failure with their lack of verifiability and transparency. 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 activists are also highly concerned that election systems will be deployed with an inappropriate pace due to irrational motivators. For example, directors of elections, secretaries of state, or legislator may make promises about election modernization to win an election or to look better than a sister state or jurisdiction. Or perhaps a technology like Internet voting is deployed, seemingly successfully, at the local level for an inconsequential election and, based upon that "success", electoral authorities decide, without any evidence that it is a wise decision, to reuse the same technology in the next federal election. These slides down the slippery slope of adoption are another major concern and must be avoided.

As such, the political and legal challenges—and related opportunities—focus on how to legislate the evidence-based measured introduction of new elections technologies, Internet voting included. *The precise formulation for a not-too-hot, not-too-cold pace, milestones, and success criteria for the introduction of E2E-VIV systems must be a primary focus on any next phase of this project.*

10.3.2 Research Challenges

The key research challenge highlighted in this report are twofold, focusing on the standard balance between security and usability.

First, the critical definition of the E2E-VIV protocol bespoke for U.S.A. elections is very challenging. In particular, the research community does not immediately know how to solve four key challenges: (i) how to handle large-scale dispute resolution, (ii) how to make verifiability comprehensible and useful to the average voter, and (iii) how to authenticate voters for public elections, and (iv) how to avoid voter coercion and vote selling in the context of digital observation of voting and verification.

Second, the usability facets of E2E-VIV also still has several research challenges. Principally, they are (v) how to ensure usable vote privacy in the presence of client-side malware and (vi) how to ensure that verification is comprehensible, usable, and accessible to the typical set of voters.

While each of these six questions is challenging, the community believes that they are surmountable for a dedicated team in a speculative next phase of this project.

10.3.3 Engineering Challenges

The engineering challenges are straightforward, at least to those organizations that have expertise in high-assurance systems. Even so, this challenge is not for the faint-hearted. After all, deploying a high-assurance distributed system of the scale and import of a public E2E-VIV election system has never been attempted.

That being said, given the size, complexity, and nature of verified software systems being deployed in military, intelligence, scientific, and civilian settings, this engineering challenge can be conquered with the right team with appropriate resources.

10.3.4 Business Opportunities

Lastly, the business opportunities for an E2E-VIV system—and the potential positive impact on the world that such a system will have—is an enormous motivator for many.

Imagine a world where inexpensive elections have high participation rates by a well-informed, engaged public. Imagine elections where the disabled and abled have equal vote and equal opportunity. Imagine elections where corrupt electoral authorities or governments have no ability to manipulate the outcome. Imagine elections that truly capture the voice of the people and increase their confidence and trust in their government.

This opportunity to impact the world for good through trustworthy democracy is a supremely worthwhile goal, and while challenges remain, we should strive toward it.

Appendix A

BON Representation of E2E-VIV Requirements (Dan/Joe K.) (50%)

BON (from external files) will appear here. Currently it is just dumped in a somewhat reasonable order, but it will be cleaned up and brought up to date.

```
scenario_chart E2EVIV_REQUIREMENTS
indexing
  title: "Requirements for End-to-end Verifiable Internet Voting Systems.";
  editor: "Joe Kiniry <kiniry@galois.com>", "Daniel M. Zimmerman <dmz@galois.com>";
  created: "16 July 2014";
  revised: "April 2015"
explanation
  "Functional and non-functional requirements for end-to-end \
 \ verifiable internet voting systems.  Requirements consisting of two \
 \ or more sentences are in fact stipulating multiple, related \
 \ requirements in a single scenario.  We index requirements from one, \
 \ thus SYSTEM_AND_DATA_ACCESS_CONTROL requirement 1 is 'Only persons \
 \ appointed by the electoral authority shall have access to the \
 \ central infrastructure, the servers and the election data.'."
end

scenario_chart TECHNICAL_REQUIREMENTS
indexing
  partof: "E2EVIV_REQUIREMENTS";
explanation
  "General technical requirements for digital elections systems."
end

scenario_chart NON_FUNCTIONAL_REQUIREMENTS
indexing
  partof: "E2EVIV_REQUIREMENTS"
explanation
  "General non-functional requirements of digital voting systems."
end

scenario_chart ACCESSIBILITY_REQUIREMENTS
indexing
  partof: "TECHNICAL_REQUIREMENTS"
explanation
  "General requirements for accessibility of digital election systems."
```

```
scenario
  "MANDATORY_ACCESSIBILITY_TESTING" -- @ref Kiniry/Zimmerman
description
  "Accessibility testing for disabled and abled voters shall be performed, \
 \ and the reports of the testing made public. The system must achieve \
 \ satisfactory accessibility testing results before being used in a \
 \ binding election."

-- @ref Rec(2004)11 Accessibility
scenario "UNIVERSAL_ACCESSIBILITY" -- @ref Rec(2004)11 Appendix III, A. 61.
description
  "Measures shall be taken to ensure that the relevant software and \
 \ services can be used by all voters and, if necessary, provide access \
 \ to alternative ways of voting."

scenario "ACCESSIBILITY_STAKEHOLDERS" -- @ref Rec(2004)11 Appendix III, A. 62.
description
  "Users shall be involved in the design of e-voting systems, \
 \ particularly to identify constraints and test ease of use at each \
 \ main stage of the development process."

scenario "USER_FACILITIES_FOR_ACCESSIBILITY" -- @ref Rec(2004)11 Appendix III, A. 63.
description
  "Users shall be supplied, whenever required and possible, with \
 \ additional facilities, such as special interfaces or other \
 \ equivalent resources, such as personal assistance."

scenario "COMPLEMENT_ACCESSIBILITY_TECHNOLOGIES" -- @ref Rec(2004)11 Appendix III, A.
    64.
description
  "Consideration shall be given, when developing new products, to \
 \ their compatibility with existing ones, including those using \
 \ technologies designed to help people with disabilities."

scenario "ACCESSIBLE_VOTING_OPTIONS" -- @ref Rec(2004)11 Appendix III, A. 65.
description
  "The presentation of the voting options shall be optimised for the \
 \ voter."
end

scenario_chart ASSURANCE_REQUIREMENTS
indexing
  partof: "NON_FUNCTIONAL_REQUIREMENTS"
explanation
  "General non-functional assurance requirements which increase system \
 \ and election assurance."

scenario
  "CLIENT_ENVIRONMENTS" -- @ref David Jefferson
description
  "Client side software (applications, apps, scripts, etc.) should be \
 \ free of known bugs on a wide range of platform and software stack \
 \ combinations intended to be usable as voting terminals."

scenario
  "AUTHENTICATION_RESILIENCE" -- @ref David Jefferson
description
  "There must be no way to automate forging or invalidation of \
 \ voter authentications without compromising the cryptographic \
```

```
 \ protocols or secrets used in the system."

scenario
  "OPEN_DOCUMENTATION" -- @ref David Jefferson
description
  "All aspects of the design, architecture, algorithms and \
 \ documentation for the entire Internet voting system (not just the \
 \ E2EV core) should be published and available for free download by \
 \ anyone."

scenario
  "DOCUMENTATION_CONSISTENCY" -- @ref David Jefferson
description
  "As the system changes, all documentation must be kept up to \
 \ date.  No new version of an E2EV Internet voting system may be \
 \ certified until all documentation is up to date."

scenario
  "OPEN_SOURCE" -- @ref David Jefferson
description
  "The source code, build scripts, issue tracking system, security \
 \ features, and related development information for the entire \
 \ Internet voting system (all versions for all platforms) shall be \
 \ made publicly available for free download and inspection by \
 \ anyone."

scenario
  "SOURCE_LICENSE" -- @ref David Jefferson
description
  "The source code for all parts of the E2EV Internet voting system \
 \ shall be made publicly available under a license that permits \
 \ anyone to download the code and build, instrument, and test it."

end

scenario_chart AUDITING_REQUIREMENTS
indexing
  partof: "TECHNICAL_REQUIREMENTS"
explanation
  "General requirements pertaining to auditing systems and digital \
 \ election systems."

-- @ref Rec(2004)11 Audit, I. General
scenario "AUDIT_SYSTEMS" -- @ref Rec(2004)11 Appendix III, E. I. 100.
description
  "The audit system shall be designed and implemented as part of the \
 \ e-voting system.  Audit facilities shall be present on different \
 \ levels of the system: logical, technical and application."

scenario "AUDITING_COMPLETENESS" -- @ref Rec(2004)11 Appendix III, E. I. 101.
description
  "End-to-end auditing of an e-voting system shall include recording, \
 \ providing monitoring facilities and providing verification \
 \ facilities."

-- @ref Rec(2004)11 Audit, II. Recording
scenario "AUDIT_SYSTEM_BASELINE" -- @ref Rec(2004)11 Appendix III, E. II. 102.
description
  "The audit system shall be open and comprehensive, and actively \
```

```
 \ report on potential issues and threats."

scenario "AUDIT_SYSTEM_DATA" -- @ref Rec(2004)11 Appendix III, E. II. 103.
description
  "The audit system shall record times, events and actions, including: \
 \ a. all voting-related information, including the number of eligible \
 \ voters, the number of votes cast, the number of invalid votes, the \
 \ counts and recounts, etc.; b. any attacks on the operation of the \
 \ e-voting system and its communications infrastructure; c. system \
 \ failures, malfunctions and other threats to the system."

-- @ref Rec(2004)11 Audit, III. Monitoring
scenario "AUDIT_SYSTEM_EVIDENCE" -- @ref Rec(2004)11 Appendix III, E. III. 104.
description
  "The audit system shall provide the ability to oversee the election \
 \ or referendum and to verify that the results and procedures are in \
 \ accordance with the applicable legal provisions."

scenario "AUDIT_DATA_SECURITY" -- @ref Rec(2004)11 Appendix III, E. IIi. 105.
description
  "Disclosure of the audit information to unauthorized persons shall \
 \ be prevented."

scenario "AUDIT_DATA_SECRECY" -- @ref Rec(2004)11 Appendix III, E. III. 106.
description
  "The audit system shall maintain voter anonymity at all times."

-- @ref Rec(2004)11 Audit, II. Verifiability
scenario "AUDIT_SYSTEM_CAPABILITY" -- @ref Rec(2004)11 Appendix III, E. IV. 107.
description
  "The audit system shall provide the ability to cross-check and \
 \ verify the correct operation of the e-voting system and the accuracy \
 \ of the result, to detect voter fraud, and to prove that all counted \
 \ votes are authentic and that all votes have been counted."

scenario "AUDIT_SYSTEM_FOR_LEGAL_COMPLIANCE" -- @ref Rec(2004)11 Appendix III, E. IV.
    108.
description
  "The audit system shall provide the ability to verify that an \
 \ e-election or e-referendum has complied with the applicable legal \
 \ provisions."

-- @ref Rec(2004)11 Audit, II. Other
scenario "AUDIT_DATA_VALIDITY" -- @ref Rec(2004)11 Appendix III, E. V. 109.
description
  "The audit system shall be protected against attacks that may \
 \ corrupt, alter or lose records in the audit system."

scenario "AUDIT_DATA_CONFIDENTIALITY" -- @ref Rec(2004)11 Appendix III, E. V. 110.
description
  "The electoral authority shall take adequate steps to ensure that the \
 \ confidentiality of any information obtained by any person while \
 \ carrying out auditing functions is guaranteed."

scenario
  "LOG_BASICS" -- @ref David Jefferson
description
  "The Internet voting system should keep detailed logs of all \
 \ relevant activity."
```

```
scenario
  "LOG_IMMUTABILITY" -- @ref David Jefferson
description
  "Log entries must be unmodifiable once written."

scenario
  "LOG_COMMITMENT" -- @ref Ron Rivest
description
  "Log entries must accurately reflect the commitment character \
 \ of elections and the relationships among election events \
 \ (e.g., ballot, vote, voter, and election state transitions)."

scenario
  "LOG_DATA_COMPLETENESS" -- @ref David Jefferson
description
  "The log data should be as complete as possible, consistent with \
 \ maximum possible vote privacy."

scenario
  "PRIVACY_VS_FRAUD_TRADEOFF" -- @ref David Jefferson
description
  "If there is a tradeoff between vote privacy and the identification \
 \ of the perpetrators of fraud, the decision should be made in favor \
 \ of vote privacy."

scenario
  "VOTER_LIST" -- @ref David Jefferson
description
  "The list of voters who voted online should be published."
end

scenario_chart AUDITING_REQUIREMENTS_VERIFICATION
indexing
  partof: "AUDITING_REQUIREMENTS"
explanation
  "Requirements specific to auditing verifiable elections."

scenario
  "VERIFICATION_PARTIAL_FAILURE" -- @ref David Jefferson
description
  "The system, in the event that it does not verify the online \
 \ votes cast, must be capable of giving an upper bound on the \
 \ number of ballots that may have been affected."

scenario
  "VERIFICATION_SOURCE" -- @ref David Jefferson
description
  "Official verification applications, like the voting software itself, \
 \ must be published in source form along with documentation, build \
 \ directions, and a standard cryptographic hash of the source code."
end

scenario_chart AUTHENTICATION_REQUIREMENTS
indexing
  partof: "TECHNICAL_REQUIREMENTS"
explanation
  "General requirements relating to the authentication of principles \
 \ (both computers and humans) involved in any digital election \
```

```
 \ system."

scenario
  "VOTER_AUTHENTICATION" -- @ref David Jefferson
description
  "The voting service must by itself securely authenticate the voter \
 \ (verify identify the voter and verify his/her registration and/or \
 \ eligibility according to law to vote in the election) before \
 \ allowing him/her to cast a ballot (or modify or replace a \
 \ previously cast ballot)."

scenario
  "NO_THIRD_PARTY_AUTHENTICATION" -- @ref David Jefferson
description
  "Authentication must not be done through third party intermediaries \
 \ such as Facebook, iCloud, Google, Yahoo, Amazon, etc. that offer \
 \ authentication services."

scenario
  "SECRET_AUTHENTICATION_SHARED_SECRETS" -- @ref David Jefferson
description
  "Authentication for remote voting systems must not use personal \
 \ information, government or commercial account identifiers, etc."

scenario
  "AUTHENTICATION_DATA_UPDATES" -- @ref David Jefferson
description
  "Authentication secrets must be changeable or revokable at \
 \ any time at the behest of either the voter or election \
 \ officials."

scenario
  "AUTHENTICATION_DATA_REFRESH_PERIODICITY" -- @ref David Jefferson
description
  "All voter authentication secrets must be changed at least once in \
 \ every election cycle."

end

scenario_chart CERTIFICATION_FUNCTIONAL_REQUIREMENTS
indexing
  partof: "TECHNICAL_REQUIREMENTS"
explanation
  "Requirements relating to the functional certification of digital \
 \ election systems and elections."

scenario "AUTOMATED_TESTING" -- @ref Kiniry/Zimmerman
description
  "Each functional requirement must have an associated set of automated \
 \ tests that provide evidence that the requirement is fulfilled."

scenario "ELECTION_PROTOCOL_PROOFS" -- @ref Kiniry/Zimmerman
description
  "The election protocol shall have associated formal proofs of correctness \
 \ and security."
end

scenario_chart CERTIFICATION_NON_FUNCTIONAL_REQUIREMENTS
indexing
```

```
  partof: "NON_FUNCTIONAL_REQUIREMENTS"
explanation
  "Requirements relating to the non-functional certification of \
 \ election systems and elections."

-- @ref Rec(2004)11 Certification
scenario "CERTIFICATION_PROCESSES" -- @ref Rec(2004)11 Appendix III, F. 111.
description
  "The electoral authority shall introduce certification processes that allow \
 \ for any ICT (Information and Communication Technology) component to \
 \ be tested and certified as being in conformity with technical \
 \ requirements."

scenario
  "CERTIFICATION_PARTIES_COMPETENCE" -- @ref David Jefferson
description
  "Any E2EV Internet voting system should be certified by competent \
 \ professionals."

scenario
  "CERTIFICATION_REPORT_TRANSPARENCY" -- @ref David Jefferson
description
  "Any and all certification reports issued by certification \
 \ professionals must be public, whether they recommend \
 \ certification or not."

scenario
  "RECERTIFICATION_CONDITIONS" -- @ref David Jefferson
description
  "Any time there is a change in the voting system client or server \
 \ side or the E2EV system, all of the requirements must \
 \ be re-established and recertified.  Changes that mandate \
 \ re-certification include, but are not limited to: new supported \
 \ hardware platforms, OS's, browsers, etc.; bug fixes and security \
 \ patches to voting client and/or server; changes or upgrades to \
 \ voting client or server in response to detected bugs or security \
 \ vulnerabilities, changes in law, or changes in threat environment."

scenario
  "RECERTIFICATION_PERIODICITY" -- @ref David Jefferson
description
  "The requirements must be re-established and recertified every \
 \ election cycle even if there are no changes."

scenario
  "VALIDATION_PLATFORM_COVERAGE" -- @ref David Jefferson
description
  "The system must be extensively tested on a wide range of platform \
 \ and software combinations."

scenario
  "PUBLIC_VALIDATION_PLATFORM_COVERAGE_RESULTS" -- @ref David Jefferson
description
  "All test procedures and results for platform coverage must be public."

end

scenario_chart EVOLVABILITY_REQUIREMENTS
indexing
```

```
  partof: "NON_FUNCTIONAL_REQUIREMENTS"
explanation
  "General requirements on the evolvability of digital election \
 \ systems."

scenario
  "ELECTORAL_AUTHORITY_UPDATE"
description
  "The electoral authority has the right and ability to update \
 \ election systems to conform to changes in applicable law, \
 \ available techology, or the system threat model."

end

scenario_chart FUNCTIONAL_REQUIREMENTS
indexing
  partof: "TECHNICAL_REQUIREMENTS"
explanation
  "General functional requirements for digital election systems."

scenario
  "CASTING_ATOMIC" -- @ref David Jefferson
description
  "Ballot casting shall be atomic with respect to server failures."

scenario
  "DETERMINISTIC_VOTING_PROCESS" -- @ref David Jefferson
description
  "If a server side failure occurs, no voter's balloting can be \
 \ left in an unknown state."

scenario
  "BALLOT_FINAL_STATES" -- @ref David Jefferson
description
  "Either a ballot is securely and completely cast and the \
 \ voter is marked as having voted, or no ballot is recorded and the \
 \ voter is not marked as having voted."

scenario
  "VOTE_RECORD_MONOTONICITY" -- @ref David Jefferson
description
  "If the system and the law allows a voter to cast multiple votes \
 \ with only the last one counting, or to cast a partial ballot with \
 \ the option of modifying it later, then each voting session must be \
 \ atomic with respect to server failures. If a failure occurs during the \
 \ voter's last session, then the votes cast as of his or her previous \
 \ session will count."

scenario
  "RECEIPT_FREEDOM" -- @ref David Jefferson
description
  "There must be no way for voters to prove to another party any \
 \ information regarding how they voted in any race (beyond what is \
 \ mathematically deducible from the final distribution of votes)."

scenario
  "VALID_BALLOT_PROVENANCE" -- @ref David Jefferson
description
  "Once it is determined that a ballot will be counted, the ballot \
```

```
 \ shall be irrevocably separated from the identification of the \
 \ voter who cast it."

scenario
  "MULTI_BALLOT_RECORD" -- @ref David Jefferson
description
  "If the voting system permits voters to modify or replace their \
 \ previously cast ballots, only the latest vote by each voter in \
 \ each race shall be counted in the final tally."

scenario
  "NO_DOUBLE_VOTE" -- @ref David Jefferson
description
  "But for systems supporting MULTI_BALLOT_RECORD, the voting system \
 \ shall not record more than one vote for any voter in any race."

scenario
  "NO_ADVERTISING" -- @ref David Jefferson
description
  "The voting system client must not display or permit the display of \
 \ any advertising or commercial logos in the window that contains the \
 \ voting session, other than those of the election jurisdiction \
 \ itself."

scenario
  "NO_EXTERNAL_LINKS" -- @ref David Jefferson
description
  "The voting system client must not display any links to other sites \
 \ except for help in the mechanics of voting."

end

scenario_chart INTEROPERABILTY_REQUIREMENTS
indexing
  partof: "TECHNICAL_REQUIREMENTS"
explanation
  "General requirements on the interoperability of digital election \
 \ systems."

-- @ref Rec(2004)11 Interoperabilty
scenario "OPEN_STANDARDS" -- @ref Rec(2004)11 Appendix III, B. 66.
description
  "Open standards shall be used to ensure that the various technical \
 \ components or services of an e-voting system, possibly derived \
 \ from a variety of sources, interoperate."

scenario "EML" -- @ref Rec(2004)11 Appendix III, B. 67.
description
  "The Election Markup Language (EML) shall be used whenever possible \
 \ for e-election and e-referendum applications."

scenario "DATA_LOCALIZATION" -- @ref Rec(2004)11 Appendix III, B. 68.
description
  "In cases that imply specific election or referendum data \
 \ requirements, a localization procedure shall be used to accommodate \
 \ these needs."

scenario
  "OPEN_LOG_FORMATS" -- @ref David Jefferson
```

```
description
  "The log data and documentation of its meaning and format shall be \
 \ available for public download so that anyone can download, inspect, \
 \ and publish concerns based on the logs."
end

scenario_chart LEGAL_REQUIREMENTS
indexing
  partof: "NON_FUNCTIONAL_REQUIREMENTS"
explanation
  "General legal requirements relating to legal matters and digital \
 \ election systems."

-- @ref Rec(2004)11 Universal Suffrage
scenario -- @ref Rec(2004)11 Appendix I, A. I. 1.
  "USABLE_UI"
description
  "The voter interface of an e-voting system shall be understandable and \
 \ easily usable."

scenario -- @ref Rec(2004)11 Appendix I, A. I. 2.
  "UNIMPEDED_REGISTRATION"
description
  "Possible registration requirements for e-voting shall not pose \
 \ an impediment to the voter participating in e-voting."

scenario -- @ref Rec(2004)11 Appendix I, A. I. 3.
  "MAXIMIZE_DISABLED_ACCESSIBILITY"
description
  "E-voting systems shall be designed, as far as it is practicable, to \
 \ maximize the opportunities that such systems can provide for persons \
 \ with disabilities."

scenario -- @ref Rec(2004)11 Appendix I, A. I. 4.
  "REMOTE_ONLY_SUPPLEMENTARY"
description
  "Unless channels of remote e-voting are universally accessible, they \
 \ shall be only an additional and optional means of voting."

-- @ref Rec(2004)11 Equal suffrage
scenario -- @ref Rec(2004)11 Appendix I, A. II. 5--6.
  "AT_MOST_ONE_VOTE_PER_VOTER"
description
  "The e-voting system shall ensure that at most one electronic vote from \
 \ each voter is included in the final tally."

scenario -- @ref Rec(2004)11 Appendix I, A. II. 7.
  "VALID_TALLY"
description
  "Every vote deposited in an electronic ballot box shall be counted, and \
 \ each vote cast in the election or referendum shall be counted only once."

scenario -- @ref Rec(2004)11 Appendix I, A. II. 8.
  "VOTE_AGGREGATION"
description
  "Where electronic and non-electronic voting channels are used in the same \
 \ election or referendum, there shall be a secure and reliable method to \
 \ aggregate all votes and to calculate the correct result."
```

```
-- @ref Rec(2004)11 Free suffrage
scenario -- @ref Rec(2004)11 Appendix I, A. III. 9.
  "FREE_SUFFRAGE"
description
  "The organization of e-voting shall secure the free formation and \
 \ expression of the voter's opinion and, where required, the \
 \ personal exercise of the right to vote."

scenario -- @ref Rec(2004)11 Appendix I, A. III. 10.
  "REFLECTIVE_VOTING_PROCESS"
description
  "The way in which voters are guided through the e-voting process \
 \ shall be such as to prevent their voting precipitately or without \
 \ reflection."

scenario -- @ref Rec(2004)11 Appendix I, A. III. 11.
  "FLEXIBLE_VOTING_PROCESS"
description
  "Voters shall be able to alter their choice at any point in the \
 \ e-voting process before casting their vote, or to break off the \
 \ procedure, without their previous choices being recorded or made \
 \ available to any other person."

scenario -- @ref Rec(2004)11 Appendix I, A. III. 12.
  "NO_VOTER_MANIPULATION"
description
  "The e-voting system shall not permit any manipulative influence to \
 \ be exercised over the voter during the voting."

scenario -- @ref Rec(2004)11 Appendix I, A. III. 13.
  "BLANK_VOTE"
description
  "The e-voting system shall provide the voter with a means of \
 \ participating in an election or referendum without the voter \
 \ exercising a preference for any of the voting options, for example, \
 \ by casting a blank vote."

scenario -- @ref Rec(2004)11 Appendix I, A. III. 14.
  "CONCLUSION_OF_VOTING_PROCESS"
description
  "The e-voting system shall indicate clearly to the voter when the \
 \ vote has been cast successfully and when the whole voting procedure \
 \ has been completed."

scenario -- @ref Rec(2004)11 Appendix I, A. III. 15.
  "IMMUTABLE_VOTES"
description
  "Except in systems supporting MULTI_BALLOT_RECORD, the e-voting system \
 \ shall prevent the changing of a vote once that vote has been cast."

-- @ref Rec(2004)11 Secret suffrage
scenario -- @ref Rec(2004)11 Appendix I, A. IV. 16.
  "SECRET_SUFFRAGE"
description
  "E-voting shall be organized in such a way as to exclude at any \
 \ stage of the voting procedure and, in particular, at voter \
 \ authentication, anything that would endanger the secrecy of the \
 \ vote."
```

```
scenario -- @ref Rec(2004)11 Appendix I, A. IV. 17.
  "ANONYMOUS_VOTES"
description
  "The e-voting system shall guarantee that votes in the electronic \
 \ ballot box and votes being counted are, and will remain, anonymous, \
 \ and that it is not possible to reconstruct a link between the vote \
 \ and the voter."

scenario -- @ref Rec(2004)11 Appendix I, A. IV. 18.
  "NO_INDIRECT_SECRECY_VIOLATION"
description
  "The e-voting system shall be so designed that the expected number \
 \ of votes in any electronic ballot box will not allow the result to \
 \ be linked to individual voters."

scenario -- @ref Rec(2004)11 Appendix I, A. IV. 19.
  "NO_SECRET_SUFFRAGE_SIDE_CHANNEL"
description
  "Measures shall be taken to ensure that the information needed \
 \ during electronic processing cannot be used to breach the secrecy of \
 \ the vote."

scenario
  "NO_NDAS_FOR_STUDY" -- @ref David Jefferson 22-6-2014
description
  "No nondisclosure agreement or any other contract shall be required \
 \ to download and study the Internet voting system."

scenario
  "NO_NDAS_FOR_AUDIT" -- @ref David Jefferson 22-6-2014
description
  "No nondisclosure agreement or any other contract shall be required \
 \ to download, instrument, build, test, and publish test results for \
 \ an E2EV Internet voting system."

end

scenario_chart MAINTENANCE_REQUIREMENTS
indexing
  partof: "NON_FUNCTIONAL_REQUIREMENTS"
explanation
  "General requirements relating to the maintainence of digital election \
 \ systems."

scenario
  "ELECTORAL_AUTHORITY_PATCH"
description
  "The electoral authority has the right and ability to patch \
 \ election systems to correct flaws discovered in the algorithms, \
 \ implementation, or deployment."

end

scenario_chart OPERATIONAL_REQUIREMENTS
indexing
  partof: "NON_FUNCTIONAL_REQUIREMENTS"
explanation
  "General operational requirements for digital election systems."
```

```
-- @ref Rec(2004)11 Notification
scenario "ELECTION_TIMETABLES" -- @ref Rec(2004)11 Appendix II, I. 36.
description
  "Domestic legal provisions governing an e-election or e-referendum \
 \ shall provide for clear timetables concerning all stages of the \
 \ election or referendum, both before and after the election or \
 \ referendum."

scenario "ELECTION_PERIOD" -- @ref Rec(2004)11 Appendix II, I. 37.
description
  "The period in which an electronic vote can be cast shall not begin \
 \ before the notification of an election or a referendum.  Particularly \
 \ with regard to remote e-voting, the period shall be defined and made \
 \ known to the public well in advance of the start of voting."

scenario "EVOTING_OUTREACH" -- @ref Rec(2004)11 Appendix II, I. 38.
description
  "The voters shall be informed, well in advance of the start of \
 \ voting, in clear and simple language, of the way in which the \
 \ e-voting will be organised, and any steps a voter may have to take \
 \ in order to participate and vote."

-- @ref Rec(2004)11 Voters
scenario "VOTER_VERIFIABLE_VOTER_REGISTER" -- @ref Rec(2004)11 Appendix II, II. 39.
description
  "There shall be a voters' register that is regularly updated.  The \
 \ voter shall be able to check, as a minimum, the information that is \
 \ held about him/her on the register, and request corrections."

scenario "ONLINE_VOTER_REGISTER" -- @ref Rec(2004)11 Appendix II, II. 40.
description
  "The possibility of creating an electronic register and introducing \
 \ a mechanism allowing online application for voter registration \
 \ and, if applicable, for application to use e-voting, shall be \
 \ considered.  If participation in e-voting requires a separate \
 \ application by the voter and/or additional steps, an electronic, \
 \ and, where possible, interactive procedure shall be considered."

scenario "VOTER_REGISTRATION_ELECTION_OVERLAP" -- @ref Rec(2004)11 Appendix II, II.
    41.
description
  "In cases where there is an overlap between the period for voter \
 \ registration and the voting period, provision for appropriate \
 \ voter authentication shall be made."

-- @ref Rec(2004)11 Candidates
scenario "ONLINE_CANDIDATE_NOMINATION" -- @ref Rec(2004)11 Appendix II, III. 42.
description
  "The possibility of introducing online candidate nomination may be \
 \ considered."

scenario "PUBLIC_CANDIDATE_LIST" -- @ref Rec(2004)11 Appendix II, III. 43.
description
  "A list of candidates that is generated and made available \
 \ electronically shall also be publicly available by other means."

-- @ref Rec(2004)11 Voting
scenario "MULTIPLE_CHANNELS_ONE_VOTE" -- @ref Rec(2004)11 Appendix II, IV. 44.
description
```

```
  "Where remote e-voting takes place while polling stations are open, \
 \ the system shall be so designed that it prevents any voter from \
 \ voting more than once."

scenario "VOTING_PERIOD_INVARIANT" -- @ref Rec(2004)11 Appendix II, IV. 45.
description
  "Remote e-voting may start and/or end at an earlier time than the \
 \ opening of any polling station.  Remote e-voting shall not continue \
 \ after the end of the voting period at polling stations."

scenario "UNIVERSAL_VOTER_HELP" -- @ref Rec(2004)11 Appendix II, IV. 46.
description
  "For every e-voting channel, support and guidance arrangements on \
 \ voting procedures shall be set up for, and be available to, the \
 \ voter.  In the case of remote e-voting, such arrangements shall also \
 \ be available through a different, widely-available communication \
 \ channel."

scenario "FAIR_VOTING_OPTIONS" -- @ref Rec(2004)11 Appendix II, IV. 47.
description
  "There shall be equality in the manner of presentation of all voting \
 \ options on the device used for casting an electronic vote."

scenario "VOTING_OPTIONS_ONLY" -- @ref Rec(2004)11 Appendix II, IV. 48.
description
  "The electronic ballot by which an electronic vote is cast shall be \
 \ free from any information about voting options, other than that \
 \ strictly required for casting the vote.  The e-voting system shall \
 \ avoid the display of other messages that may influence the voters' \
 \ choice."

scenario "FAIR_VOTING_OPTION_INFORMATION" -- @ref Rec(2004)11 Appendix II, IV. 49.
description
  "If it is decided that information about voting options will be \
 \ accessible from the e-voting site, this information shall be \
 \ presented with equality."

scenario "BINDING_ELECTION_CLARITY" -- @ref Rec(2004)11 Appendix II, IV. 50.
description
  "Before casting a vote using a remote e-voting system, voters' \
 \ attention shall be explicitly drawn to the fact that the e-election \
 \ or e-referendum in which they are submitting their decision by \
 \ electronic means is a real election or referendum.  In case of \
 \ tests, participants shall have their attention drawn explicitly to \
 \ the fact that they are not participating in a real election or \
 \ referendum and shall, when tests are continued at election times, \
 \ at the same time be invited to cast their ballot by the voting \
 \ channel(s) available for that purpose."

scenario "REMOTE_RECEIPT_FREEDOM" -- @ref Rec(2004)11 Appendix II, IV. 51.
description
  "A remote e-voting system shall not enable the voter to be in \
 \ possession of a proof of the content of the vote cast."

scenario "SUPERVISED_VOTE_RECEIPT_FREEDOM" -- @ref Rec(2004)11 Appendix II, IV. 52.
description
  "In a supervised environment, the information on the vote shall \
 \ disappear from the visual, audio or tactile display used by the \
 \ voter to cast the vote as soon as it has been cast.  Where a paper \
```

```
 \ proof of the electronic vote is provided to the voter at a polling \
 \ station, the voter shall not be able to show it to any other per- \
 \ son, or take this proof outside of the polling station."

-- @ref Rec(2004)11 Results
scenario "SECRET_INTERMEDIATE_TALLY" -- @ref Rec(2004)11 Appendix II, V. 53.
description
  "The e-voting system shall not allow the disclosure of the number of \
 \ votes cast for any voting option until after the closure of the \
 \ electronic ballot box.  This information shall not be disclosed to \
 \ the public until after the end of the voting period."

scenario "NO_ITALIAN_ATTACK" -- @ref Rec(2004)11 Appendix II, V. 54.
description
  "The e-voting system shall prevent processing information on votes \
 \ cast within deliberately chosen sub-units that could reveal \
 \ individual voters' choices."

scenario "DECODING_LATENCY" -- @ref Rec(2004)11 Appendix II, V. 55.
description
  "Any decoding required for the counting of the votes shall be \
 \ carried out as soon as practicable after the closure of the voting \
 \ period."

scenario "TALLY_OBSERVATION" -- @ref Rec(2004)11 Appendix II, V. 56.
description
  "When counting the votes, representatives of the competent electoral \
 \ authority shall be able to participate in, and any observers able to \
 \ observe, the count."

scenario "TALLY_RECORD" -- @ref Rec(2004)11 Appendix II, V. 57.
description
  "A record of the counting process of the electronic votes shall be \
 \ kept, including information about the start and end of, and the \
 \ persons involved in, the count."

scenario "INTEGRITY_VIOLATION_RECORD" -- @ref Rec(2004)11 Appendix II, V. 58.
description
  "In the event of any irregularity affecting the integrity of votes, \
 \ the affected votes shall be recorded as having their integrity violated."

-- @ref Rec(2004)11 Audit
scenario "SYSTEM_AUDITABILITY" -- @ref Rec(2004)11 Appendix II, VI. 59.
description
  "The e-voting system shall be auditable."

scenario "SYSTEM_AUDITS_IMPACT" -- @ref Rec(2004)11 Appendix II, VI. 60.
description
  "The conclusions drawn from the audit process shall be applied in \
 \ future elections and referenda."

scenario
  "OPEN_SYSTEM" -- @ref David Jefferson
description
  "The e-voting system must function correctly as an open system, \
 \ where large parts (the mix of client hardware and software in \
 \ fact) are unknown, unsecured, uncertified, and completely out \
 \ of control of election officials."
```

```
scenario
  "SUPPORTED_CLIENTS" -- @ref David Jefferson
description
  "Operators of voting systems must document exactly what client \
 \ configurations are required or supported, including: \
 \  - versions of hardware platforms (PCs, mobile devices, etc.) \
 \  - versions of specific operating systems for those platforms \
 \  - versions of specific browsers, plugins, protocols, or \
 \    other software applications, apps, components, and plugins."

scenario
  "CLIENT_INTERFERENCE" -- @ref David Jefferson
description
  "Operators of voting systems must document exactly which common \
 \ components, plugins, or other software interfere with voting (e.g., \
 \ flash blockers, popup blockers, script blockers, etc.)."

scenario
  "MANDATORY_CLIENT_TECHNOLOGY" -- @ref David Jefferson
description
  "Operators of voting systems must document exactly what configuration \
 \ choices the voter must make to successfully vote (e.g., mandate \
 \ Javascript)."

scenario
  "PRIVACY_ENHANCING_VOTER_OPTIONS" -- @ref David Jefferson
description
  "Operators of voting systems must document exactly what configuration \
 \ choices the voter might wish to make to more strongly protect \
 \ his/her vote privacy; e.g., disable cookies, run privacy-protecting \
 \ browser plugins, vote from virtual machine that is later destroyed, \
 \ log out of social networks, disable remote control and remote \
 \ administration tools, disable incoming connections, etc."

scenario
  "BREADCRUMBS_USER_ADVICE" -- @ref David Jefferson
description
  "Users may be advised to turn off browser history data, cookies, \
 \ logging data, and other tools that might retain a record of the \
 \ vote transaction whether the vote data itself or metadata."

end

scenario_chart PROCEDURAL_REQUIREMENTS
indexing
  partof: "NON_FUNCTIONAL_REQUIREMENTS"
explanation
  "General procedural requirements for digital electoin systems."

-- @ref Rec(2004)11 Transparency
scenario "VOTER_COMPREHENSION_AND_CONFIDENCE" -- @ref Rec(2004)11 Appendix I, B. I.
    20.
description
  "The electoral authority shall take steps to ensure that voters understand and \
 \ have confidence in the e-voting system in use."

scenario "PUBLIC_SYSTEM_FUNCTION" -- @ref Rec(2004)11 Appendix I, B. I. 21.
description
  "Information on the functioning of an e-voting system shall be made \
```

```
 \ publicly available."

scenario "VOTER_PRACTICE" -- @ref Rec(2004)11 Appendix I, B. I. 22.
description
  "Voters shall be provided with an opportunity to practice any new \
 \ method of e-voting before, and separately from, the moment of \
 \ casting an electronic vote."

scenario "OBSERVER_INVARIANTS" -- @ref Rec(2004)11 Appendix I, B. I. 23.
description
  "Any observers, to the extent permitted by law, shall be able to be \
 \ present to observe and comment on the e-elections, including the \
 \ establishing of the results."

-- @ref Rec(2004)11 Verifiability and accountability
scenario "DISCLOSURE_OBLIGATIONS" -- @ref Rec(2004)11 Appendix I, B. II. 24.
description
  "The components of the e-voting system shall be disclosed, at least \
 \ to the competent electoral authorities, as required for verification \
 \ and certification purposes."

scenario "CERTIFICATION_OBLIGATIONS" -- @ref Rec(2004)11 Appendix I, B. II. 25.
description
  "Before any e-voting system is introduced, and at appropriate \
 \ intervals thereafter, and in particular after any changes are made \
 \ to the system, an independent body, appointed by the electoral \
 \ authorities, shall verify that the e-voting system is working \
 \ correctly and that all the necessary security measures have been \
 \ taken."

scenario "RECOUNT_SUPPORTED" -- @ref Rec(2004)11 Appendix I, B. II. 26.
description
  "There shall be the possibility for a recount.  Other features of the \
 \ e-voting system that may influence the correctness of the results \
 \ shall be verifiable."

scenario "RERUN_SUPPORTED" -- @ref Rec(2004)11 Appendix I, B. II. 27.
description
  "The e-voting system shall not prevent the partial or complete \
 \ re-run of an election or a referendum."

-- @ref Rec(2004)11 Reliability and security
scenario "RELIABILITY_AND_SECURITY" -- @ref Rec(2004)11 Appendix I, B. III. 28.
description
  "The electoral authority shall ensure the reliability and \
 \ security of the e-voting system."

scenario "NO_FRAUD_OR_INTERVENTION" -- @ref Rec(2004)11 Appendix I, B. III. 29.
description
  "All possible steps shall be taken to avoid the possibility of fraud \
 \ or unauthorized intervention affecting the system during the whole \
 \ voting process."

scenario "SYSTEM_AVAILABILITY" -- @ref Rec(2004)11 Appendix I, B. III. 30.
description
  "The e-voting system shall contain measures to preserve the \
 \ availability of its services during the e-voting process.  It shall \
 \ resist, in particular, malfunction, breakdowns or denial of service \
 \ attacks."
```

```
scenario "SYSTEM_GENUINE_AND_CORRECT" -- @ref Rec(2004)11 Appendix I, B. III. 31.
description
  "Before any e-election or e-referendum takes place, the competent \
 \ electoral authority shall satisfy itself that the e-voting system \
 \ is genuine and operates correctly."

scenario "SYSTEM_AND_DATA_ACCESS_CONTROL" -- @ref Rec(2004)11 Appendix I, B. III. 32.
description
  "Only persons appointed by the electoral authority shall have access \
 \ to the central infrastructure, the servers and the election \
 \ data.  There shall be clear rules established for such \
 \ appointments.  Critical technical activities shall be carried out by \
 \ teams of at least two people.  The composition of the teams shall be \
 \ regularly changed.  As far as possible, such activities shall be \
 \ carried out outside election periods."

scenario "OPEN_BALLOT_BOX_INVARIANTS" -- @ref Rec(2004)11 Appendix I, B. III. 33.
description
  "While an electronic ballot box is open, any authorised intervention \
 \ affecting the system shall be carried out by teams of at least two \
 \ people, be the subject of a report, and be monitored by \
 \ representatives of the competent electoral authority and any \
 \ election observers."

scenario "VOTES_INVARIANTS" -- @ref Rec(2004)11 Appendix I, B. III. 34.
description
  "The e-voting system shall maintain the availability and integrity \
 \ of the votes.  It shall also maintain the confidentiality of the \
 \ votes and keep them sealed until the counting process.  If stored or \
 \ communicated outside controlled environments, the votes shall be \
 \ encrypted."

scenario "SEALED_VOTES_VOTER_RELATION" -- @ref Rec(2004)11 Appendix I, B. III. 35.
description
  "Votes and voter information shall remain sealed as long as the data \
 \ is held in a manner where they can be associated.  Authentication \
 \ information shall be separated from the voter's decision at a \
 \ pre-defined stage in the e-election or e-referendum."

scenario
  "VERIFICATION_FAILURE_PROCEDURES" -- @ref David Jefferson
description
  "There must be clear technical and legal procedures for how to \
 \ proceed in the event that voters can prove that their votes were \
 \ not received accurately or counted, or if the official election \
 \ verification application does not verify that the Internet part of \
 \ the election was correct."

end

scenario_chart SYSTEM_OPERATIONAL_REQUIREMENTS
indexing
  partof: "TECHNICAL_REQUIREMENTS"
explanation
  "General system operational requirements for digital election \
 \ systems."

-- @ref Rec(2004)11 Systems Operation
```

```
scenario "PUBLIC_SYSTEM_MANIFEST" -- @ref derived from Rec(2004)11 Appendix III, C.
    69.
description
  "The electoral authority shall publish an official manifest of the \
 \ software used in an e-election or e-referendum.  At the very least \
 \ the manifest shall indicate the software used, the versions, its date \
 \ of installation and a brief description. A procedure shall be established \
 \ for updating the manifest to reflect changes to the installed software."

scenario "PRIVATE_SYSTEM_MANIFEST" -- @ref derived from Rec(2004)11 Appendix III, C.
    69.
description
  "The electoral authority shall maintain a manifest of all software, \
 \ including data protection software, used in the system. This manifest \
 \ shall contain at least the same information as the public manifest. \
 \ A procedure shall be established for updating the manifest to reflect \
 \ changes to the installed software."

scenario "MANIFEST_ACCURACY" -- @ref derived from Rec(2004)11 Appendix III, C. 69.
description
  "It shall be possible for the electoral authority to check the installed \
 \ software against the system manifests at any time."

scenario "SYSTEM_FAILOVER_INVARIANTS" -- @ref Rec(2004)11 Appendix III, C. 70.
description
  "Those responsible for operating the equipment shall draw up a \
 \ contingency procedure for system failures.  Any backup system shall \
 \ conform to the same standards and requirements as the original system."

scenario "DATA_BACKUP_INVARIANTS" -- @ref Rec(2004)11 Appendix III, C. 71.
description
  "Sufficient backup arrangements shall be in place and be permanently \
 \ available to ensure that voting proceeds smoothly.  The staff \
 \ concerned shall be ready to intervene rapidly according to a \
 \ procedure drawn up by the electoral authority."

scenario "SYSTEM_INVARIANTS_DURING_ELECTION" -- @ref Rec(2004)11 Appendix III, C. 72.
description
  "Those responsible for the equipment shall use special procedures to \
 \ ensure that during the polling period the voting equipment and its \
 \ use satisfy requirements.  The backup services shall be regularly \
 \ monitored."

scenario "PRE_ELECTION_CERTIFICATION_INVARIANTS" -- @ref Rec(2004)11 Appendix III, C.
    73.
description
  "Before each election or referendum, the equipment shall be checked \
 \ and approved in accordance with a protocol drawn up by the \
 \ electoral authority.  The equipment shall be checked to ensure that \
 \ it complies with technical specifications.  The findings shall be \
 \ submitted to the electoral authority."

scenario "FORMAL_CONTROL_PROCEDURE" -- @ref Rec(2004)11 Appendix III, C. 74.
description
  "All technical operations shall be subject to a formal control \
 \ procedure.  Any substantial changes to key equipment shall be \
 \ performed with advance notice."
```

```
scenario "PHYSICAL_SECURITY_OF_SYSTEMS_INVARIANTS" -- @ref Rec(2004)11 Appendix III, C
    . 75.
description
  "Key e-election or e-referendum equipment shall be located in a \
 \ secure area and that area shall, throughout the election or \
 \ referendum period, be guarded against interference of any sort and \
 \ from any person.  During the election or referendum period a \
 \ physical disaster recovery plan shall be in place.  Furthermore, any \
 \ data retained after the election or referendum period shall be \
 \ stored securely."

scenario "INCIDENT_RESPONSE_INVARIANTS" -- @ref Rec(2004)11 Appendix III, C. 76.
description
  "Where incidents that could threaten the integrity of the system \
 \ occur, those responsible for operating the equipment shall \
 \ immediately inform the electoral authority, which will \
 \ take the necessary steps to mitigate the effects of the \
 \ incident.  The level of incident that shall be reported shall be \
 \ specified in advance by the electoral authority."

scenario "OPERATIONAL_TRANSPARENCY" -- @ref Kiniry/Zimmerman
description
  "A report containing every manifest change, every data or system \
 \ invariant violation, every control procedure violation, and every \
 \ physical security violation shall be prepared and made public by \
 \ the electoral authority after every election."

end

scenario_chart RELIABILITY_REQUIREMENTS
indexing
  partof: "SYSTEM_OPERATIONAL_REQUIREMENTS"
explanation
  "General reliability requirements for any internet election system."

scenario
  "GENERAL_MTBF" -- @ref David Jefferson
description
  "The entire voting service (server side) must have a proven MTBF of \
 \ >168 hours (1 week) under peak expected voting loads the entire \
 \ time."

scenario
  "LIVE_ELECTION_MTBF" -- @ref David Jefferson
description
  "MTBF validation must be demonstrated in multiple tests of \
 \ actual mock elections."

scenario
  "MTBF_CONTRA_DDOS" -- @ref David Jefferson
description
  "MTBF requirements apply only during normal peak operation, not \
 \ during attacks (e.g., DDoS)."

scenario
  "SYSTEM_RECOVERY_TIME" -- @ref David Jefferson
description
  "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."

scenario
  "UPTIME" -- @ref David Jefferson
description
  "The system must have three nines (99.9%) uptime."

scenario
  "FAILURE_VALIDATION" -- @ref David Jefferson
description
  "Uptime must be demonstrated by failures in actual mock election \
 \ situations, e.g. tested by sudden loss of power to any server."

scenario
  "MIRRORED_FAILOVER_SERVICE" -- @ref David Jefferson
description
  "The system must have a warm spare in a second data center that can take \
 \ over in case of major failure."

scenario
  "FAILOVER_STAFFING" -- @ref David Jefferson
description
  "The system must be staffed at all times to guarantee the 10 minute \
 \ recovery time."

scenario
  "OPERATION_UNDER_DDOS" -- @ref David Jefferson
description
  "In a federal election the voting system must remain available even \
 \ during a large distributed denial of service attack.  It must be \
 \ able to continue correct operation during a sustained DDoS attack \
 \ on any combination of server side IP addresses (whether at the \
 \ primary server data center or its ISP) at a total level of 100 Gb/s \
 \ with no more than 15s degradation of response time to voters during \
 \ the attack."

scenario
  "DDOS_REFRESH_PERIODICITY" -- @ref David Jefferson
description
  "The DDoS threshold (initially 100 Gb/s) should be evaluated every \
 \ election cycle to see if it has to be raised due to newer \
 \ DDoS attack technologies."

scenario
  "DDOS_ATTACK_VALIDATION" -- @ref David Jefferson
description
  "The ability to survive a DDoS attack must be actually demonstrated \
 \ in the actual network configuration to be used prior to each \
 \ federal election."

scenario
  "DDOS_LOCAL_ELECTION" -- @ref David Jefferson
description
  "Reduced DDoS defense requirements might be acceptable for \
 \ non-federal elections."

end

scenario_chart SECURITY_REQUIREMENTS
```

```
indexing
  partof: "TECHNICAL_REQUIREMENTS";
explanation
  "General security requirements for digital elections systems."

-- @ref Rec(2004)11 Security, I. General requirements
scenario "NO_DATA_LOSS" -- @ref Rec(2004)11 Appendix III, D. I. 77.
description
  "Technical and organizational measures shall be taken to ensure that \
 \ no data will be permanently lost in the event of a breakdown or a \
 \ fault affecting the e-voting system."

scenario "VOTER_PRIVACY_INVARIANTS" -- @ref Rec(2004)11 Appendix III, D. I. 78.
description
  "The e-voting system shall maintain the privacy of \
 \ individuals.  Confidentiality of voters' registers stored in or \
 \ communicated by the e-voting system shall be maintained."

scenario "SYSTEM_SELF_CHECKS" -- @ref Rec(2004)11 Appendix III, D. I. 79.
description
  "The e-voting system shall perform regular checks to ensure that its \
 \ components operate in accordance with its technical specifications \
 \ and that its services are available."

scenario "SYSTEM_ACCESS_CONTROL" -- @ref Rec(2004)11 Appendix III, D. I. 80.
description
  "The e-voting system shall restrict access to its services, \
 \ depending on the user identity or the user role, to those services \
 \ explicitly assigned to this user or role.  User authentication shall \
 \ be effective before any action can be carried out."

scenario "DATA_PROTECTION" -- @ref Rec(2004)11 Appendix III, D. I. 81.
description
  "The e-voting system shall protect authentication data so that \
 \ unauthorized entities cannot misuse, intercept, modify, or otherwise \
 \ gain knowledge of any of this data.  In uncontrolled \
 \ environments, authentication based on cryptographic mechanisms is \
 \ advisable."

scenario "UNIQUE_IDENTIFICATION" -- @ref Rec(2004)11 Appendix III, D. I. 82.
description
  "Identification of voters and candidates in a way that they can \
 \ unmistakably be distinguished from other persons (unique \
 \ identification) shall be ensured."

scenario "OBSERVATION_DATA" -- @ref Rec(2004)11 Appendix III, D. I. 83.
description
  "E-voting systems shall generate reliable and sufficiently detailed \
 \ observation data so that election observation can be carried \
 \ out.  The time at which an event generated observation data shall be \
 \ reliably determinable.  The authenticity, availability and \
 \ integrity of the data shall be maintained."

scenario "TIME_SYNCHRONIZATION" -- @ref Rec(2004)11 Appendix III, D. I. 84.
description
  "The e-voting system shall maintain reliable synchronized time \
 \ sources.  The accuracy of the time sources shall be sufficient to \
 \ maintain time marks for audit trails and observations data, as well \
 \ as for maintaining the time limits for registration, nomination, \
```

```
 \ voting, or counting."

scenario "SECURITY_COMPLIANCE_RESPONSIBILITY" -- @ref Rec(2004)11 Appendix III, D. I.
    85.
description
  "The electoral authority has overall responsibility for compliance \
 \ with these security requirements, and such compliance shall be assessed by \
 \ independent bodies."

-- @ref Rec(2004)11 Security, II. Requirements in pre-voting stages
scenario "LISTS_INVARIANTS" -- @ref Rec(2004)11 Appendix III, D. II. 86.
description
  "The authenticity, availability and integrity of the voters' \
 \ registers and lists of candidates shall be maintained.  The source of \
 \ the data shall be authenticated.  Provisions on data protection shall \
 \ be respected."

scenario "CANDIDATE_PROCESS_TIME_PROVENANCE" -- @ref Rec(2004)11 Appendix III, D. II.
    87.
description
  "The fact that candidate nomination and, if required, the decision \
 \ of the candidate and/or the electoral authority to accept a \
 \ nomination has happened within the prescribed time limits shall be \
 \ ascertainable."

scenario "VOTER_PROCESS_TIME_PROVENANCE" -- @ref Rec(2004)11 Appendix III, D. II. 88.
description
  "The fact that voter registration has happened within the prescribed \
 \ time limits shall be ascertainable."

-- @ref Rec(2004)11 Security, III. Requirements in the voting stage
scenario "ELECTION_DATA_INTEGRITY_INVARIANTS" -- @ref Rec(2004)11 Appendix III, D. III
    . 89.
description
  "The integrity of data communicated from the pre-voting stage \
 \ (e.g., voters' registers and lists of candidates) shall be \
 \ maintained.  Data-origin authentication shall be carried out."

scenario "BALLOT_AUTHENTICITY_INVARIANTS" -- @ref Rec(2004)11 Appendix III, D. III.
    90.
description
  "It shall be ensured that the e-voting system presents an authentic \
 \ ballot to the voter.  In the case of remote e-voting, the voter shall \
 \ be informed about the means to verify that a connection to the \
 \ official server has been established and that the authentic ballot \
 \ has been presented."

scenario "CAST_VOTE_TIME_PROVENANCE" -- @ref Rec(2004)11 Appendix III, D. III. 91.
description
  "The fact that a vote has been cast within the prescribed time \
 \ limits shall be ascertainable."

scenario "CONTROLLED_SYSTEMS_AND_VOTE_INTEGRITY" -- @design derived from Rec(2004)11
    Appendix III, D. III. 92.
description
  "Election equipment under the control of the electoral authority \
 \ shall be protected against influence that could modify the vote."

scenario "UNCONTROLLED_SYSTEMS_AND_VOTE_INTEGRITY" -- @ref Kiniry/Zimmerman
```

```
description
  "The integrity of the vote must not depend on the security of election \
 \ equipment not under the control of the electoral authority."

scenario "NO_BREADCRUMBS" -- @ref Rec(2004)11 Appendix III, D. III. 93.
description
  "Residual information holding the voter's decision or the display of \
 \ the voter's choice shall be destroyed after the vote has been \
 \ cast.  In the case of remote e-voting, the voter shall be provided \
 \ with information on how to delete, where that is possible, traces \
 \ of the vote from the device used to cast the vote."

scenario "ELIGIBILITY_IMPLIES_VOTE_VOTER_INVARIANTS" -- @ref Rec(2004)11 Appendix III,
    D. III. 94.
description
  "The e-voting system shall at first ensure that a user who tries to \
 \ vote is eligible to vote.  The e-voting system shall authenticate \
 \ the voter and shall ensure that only the appropriate number of votes \
 \ per voter is cast and stored in the electronic ballot box."

scenario "VOTE_CHOICE_INVARIANTS" -- @ref Rec(2004)11 Appendix III, D. III. 95.
description
  "The e-voting system shall ensure that the voter's choice is \
 \ accurately represented in the vote and that the sealed vote enters \
 \ the electronic ballot box."

scenario "END_OF_VOTE_PERIOD_INVARIANTS" -- @ref Rec(2004)11 Appendix III, D. III. 96.
description
  "After the end of the e-voting period, no voter shall be allowed to \
 \ gain access to the e-voting system.  However, the acceptance of \
 \ electronic votes into the electronic ballot box shall remain open \
 \ for a sufficient period of time to allow for any delays in the \
 \ passing of messages over the e-voting channel."

-- @ref Rec(2004)11 Security, IV. Requirements in post-voting stages
scenario "DATA_COMMUNICATION_INTEGRITY_INVARIANTS" -- @ref Rec(2004)11 Appendix III, D
    . IV. 97.
description
  "The integrity of data communicated during the voting stage \
 \ (e.g. votes, voters' registers, lists of candidates) shall be \
 \ maintained.  Data-origin authentication shall be carried out."

scenario "TALLY_INVARIANTS" -- @ref Rec(2004)11 Appendix III, D. IV. 98.
description
  "The counting process shall accurately count the votes.  The counting \
 \ of votes shall be reproducible."

scenario "BALLOT_BOX_AND_TALLY_INVARIANTS" -- @ref Rec(2004)11 Appendix III, D. IV.
    99.
description
  "The e-voting system shall maintain the availability and integrity \
 \ of the electronic ballot box and the output of the counting process \
 \ as long as required."

scenario "ADVERSARY_RESOURCES" -- @ref Kiniry/Zimmerman
description
  "The e-voting system shall be designed and tested with the assumption \
 \ that an adversary has a budget of $10 per voter per election, which they \
 \ can apply toward any critical subset of votes/voters of their choosing."
```

```
end

scenario_chart E2EVIV_SECURITY_REQUIREMENTS
indexing
  partof: "SECURITY_REQUIREMENTS";
  author: "David Jefferson <d_jefferson@yahoo.com>";
  created: "22 June 2014";
  reviewer: "Joe Kiniry <kiniry@galois.com>";
  reviewed: "16 July 2014"
explanation
  "General security requirements for end-to-end verifiable internet \
 \ election systems."

 -- These are requirements for embedding an E2EV system in an Internet
 -- voting environment.  They are over and above the requirements for
 -- the core E2EV itself.  We do not consider usability or accessibility
 -- requirements here.  Some of these requirements will make
 -- accessibility and usability more difficult to achieve.  Still, these
 -- are requirements, and if they cannot be met, or cannot be met
 -- simultaneously with usability and accessibility requirement, then we
 -- have to recommend not implementing an E2EV Internet voting system.

scenario
  "NATIONAL_SECURITY" -- @ref David Jefferson
description
  "If used in federal elections, an Internet voting system is also a \
 \ national security system, and thus must be subject to the highest \
 \ security requirements."

scenario
  "FEDERAL_REQUIREMENTS" -- @ref David Jefferson
description
  "Any Internet voting system used in a public primary or general \
 \ election in the U.S. for federal or state legislative, executive, \
 \ or judicial office, or recall election, or statewide initiative or \
 \ referendum, must meet all of the requirements in this document."

scenario
  "LOCAL_REQUIREMENTS" -- @ref David Jefferson
description
  "Reduced security requirements might be appropriate for county, \
 \ municipal, or other kinds of elections"

scenario
  "AUTOMATED_REGISTRATION_FRAUD" -- @ref David Jefferson
description
  "Automated registration fraud must not be possible."
 -- Eligibility & Registration (online registration, automated
 -- registration fraud, and change of credentials): DJ doesn't know yet
 -- what to write here regarding requirements.  But obviously any
 -- automated registration fraud can be used to affect the outcome of
 -- elections.

scenario
  "CLIENT_SIDE_AUTHENTICITY" -- @ref David Jefferson
description
  "There must be a means by which any third party can determine if the \
 \ client-side software is genuine."
 -- Authentication of service: Not sure what requirement should be
```

```
 -- here.  The intent is to somehow ascertain that the E2EV software
 -- on the client-side is genuine.  Presumably that E2EV software will
 -- authenticate the remote server.

scenario
  "AUTHENTICATION_INDEPENDENCE" -- @ref David Jefferson
description
  "The security of authentication must not be affected by \
 \ any potential breach of any public or commercial databases."

scenario
  "ZERO_KNOWLEDGE_AUTHENTICATION" -- @ref David Jefferson
description
  "It should not be possible for an attacker to impersonate voters \
 \ even if the entire server database used for authentication is \
 \ compromised."

scenario
  "AUTHENTICATION_CREDENTIAL_REESTABLISHMENT" -- @ref David Jefferson
description
  "In some cases of security breach it must be possible to require all \
 \ voters in a jurisdiction to re-establish credentials."
end

scenario_chart PRIVACY_REQUIREMENTS
indexing
  partof: "SECURITY_REQUIREMENTS"
explanation
  "General privacy requirements for end-to-end verifiable internet \
 \ election systems."
-- violations of vote privacy are not generally detectable
-- violations of vote privacy are irreversible
-- violations of vote privacy enable vote coercion and vote selling
-- vote privacy cannot be verified by testing; it can only be ascertained by expert
    analysis of architecture and code

scenario
  "E2E_VOTE_PRIVACY" -- @ref David Jefferson
description
  "Vote privacy must be preserved end-to-end insofar as mathematically \
 \ possible."

scenario
  "VOTE_PRIVACY_INVIOLATE" -- @ref David Jefferson
description
  "Vote privacy cannot be waived by voters."

scenario
  "MALWARE_PRESENCE" -- @ref David Jefferson
description
  "Vote privacy must not be violated even in the presence of arbitrary \
 \ malicious code on the client platform, including phony client \
 \ software, malicious client wrappers, MITM code between the user and \
 \ the E2EV interface, malicious browser plugins or scripts, \
 \ keyloggers, etc."
 -- This requirement will seriously complicate the user interface an
 -- usability of the system, but is absolutely essential.

scenario
```

```
  "REMOTE_MONITORING" -- @ref David Jefferson
description
  "Voting should not be permitted from client platforms known to have \
 \ remote monitoring software installed that could be used to monitor \
 \ or log voting activity and that cannot be turned off by the voter. \
 \ (All mobile platforms had, and probably still do have, such remote \
 \ monitoring software.)"

scenario
  "CLIENT_SIDE_CHANNELS" -- @ref David Jefferson
description
  "The client software of the voting system must not send data to any \
 \ IP address except those associated with the vote server and the \
 \ basic infrastructure servers of the Internet."

scenario
  "SOCIAL_MEDIA_SIDE_CHANNELS" -- @ref David Jefferson
description
  "The client should not provide any information to third parties, \
 \ e.g., Facebook, Twitter, etc. regarding the act of voting."

scenario
  "NO_TRACKING" -- @ref David Jefferson
description
  "There must be no tracking devices or tracking logic in the vote \
 \ client."

scenario
  "NO_BREADCRUMBS_DETAILS" -- @ref David Jefferson
description
  "The client software must leave no files or other persistent data on \
 \ the platform regarding the vote transaction but for an optional \
 \ file containing information needed for subsequent verification that \
 \ the voter's ballot is included in the election canvass: no cookies \
 \ or other session files, no temporary files."

scenario
  "TRANSIENT_DATA_CLEANUP" -- @ref David Jefferson
description
  "The client software should explicitly erase (i.e., overwrite) all \
 \ transient copies of vote-transaction data, e.g. data in registers, \
 \ caches, RAM, and virtual memory."

scenario
  "FORENSICALLY_SECURE" -- @ref David Jefferson
description
  "It should not be possible even for client-side forensic tools to \
 \ retrieve any information regarding the voting transaction after the \
 \ voting session is ended."

scenario
  "REMOTE_ADMINISTRATION_FORBIDDEN" -- @ref David Jefferson
description
  "The voting system should not support platforms that have remote \
 \ administration or remote control tools installed that cannot be \
 \ turned off by the voter."

scenario
  "INVULNERABLE_TO_ELECTION_MALWARE" -- @ref David Jefferson
```

```
description
  "The voting system must not be vulnerable to malware designed to \
 \ modify votes before they are input to the E2EV system."
 -- This will seriously complicate the human interface and usability
 -- of the voting system, but is absolutely essential.  Malware can be
 -- in many forms: completely phony or "alternative" client app,
 -- client wrapper, client-side MITM, browser plugin, client APT, etc.

scenario
  "CLIENT_SYSTEM_AUTHENTICATION" -- @ref David Jefferson
description
  "The voting system server must authenticate that it is communicating \
 \ with a genuine vote client during a voting session."
 -- This will complicate, but not eliminate, the possibility of
 -- client-side malware.  @see CLIENT_SIDE_AUTHENTICITY.

scenario
  "PENETRATION_ATTACKS" -- @ref David Jefferson
description
  "Deny penetration attacks. (DJ doesn't know what to write about \
 \ this.)"

scenario
  "APT_ATTACKS" -- @ref David Jefferson
description
  "Deny advanced persistent threat attacks.  (DJ doesn't know what to \
 \ write about this.)"

scenario
  "INSIDER_ATTACKS" -- @ref David Jefferson
description
  "Something about insider attacks being impossible.  (DJ doesn't know \
 \ what to write about this.)"

scenario
  "COERCION_PREVENTION" -- @ref David Jefferson
description
  "There must be no way for voters to prove to another party any \
 \ information regarding how they voted in any race beyond what is \
 \ mathematically deducible from the final distribution of votes."
  -- @see RECEIPT_FREEDOM

scenario
  "SOFTWARE_INDEPENDENCE" -- @ref Ron Rivest
description
  "The system must witness software independence: the tabulation \
 \ record must not rely solely on software."

scenario
  "DIGITAL_EVIDENCE_NOT_A_RECEIPT"
description
  "Digital evidence (e.g., photographing a ballot or video recording \
 \ the casting process) of the voting process must not violate receipt \
 \ freedom."
end

scenario_chart CERTIFICATION_AND_RECERTIFICATION_REQUIREMENTS
indexing
  partof: "SECURITY_REQUIREMENTS"
```

```
explanation
  "General security requirements relating to certification of digital \
 \ elections systems."
end

scenario_chart USABILITY_REQUIREMENTS
indexing
  partof: "TECHNICAL_REQUIREMENTS"
explanation
  "General usability requirements of digital elections systems."

scenario
  "MANDATORY_USABILITY_TESTING" -- @ref Kiniry/Zimmerman
description
  "Usability testing for disabled and abled voters shall be performed, \
 \ and the reports of the testing made public. The system must achieve \
 \ satisfactory usability testing results before being used in a \
 \ binding election."

scenario
  "VOTE_CONFIRMATION" -- @ref David Jefferson
description
  "If a voter receives the final 'Thank you for voting' confirmation, \
 \ then she/he can be certain the ballot was recorded."

scenario
  "UNCERTAIN_VOTER_REVOTE" -- @ref David Jefferson
description
  "If the voter is uncertain about the state of their ballot, he/she \
 \ is free to attempt to vote again."

end
```

Appendix B

Expert Statements (Dan/Joe K.) (0%)

B.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.

1. Public elections should not be conducted over the Internet using systems that are not end-to-end verifiable.

2. 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.

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.

[1] Remote voting is defined here as voting without the benefit of the public monitoring that takes place in a traditional poll site.

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.

Appendix C

Usability Study Report (Keith/Judy) (100%)

The copy editor needs to drop in the text of the report and we'll cite the original PDF version as well, which will be provided as a separately downloadable artifact on the project website. - JRK

Bibliography

[1] *2012 Election Administration and Voting Survey*. 2013.

[2] T. Abdoul et al. "AADL Execution Semantics Transformation for Formal Verification". In: *13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2008)*. 2008, pp. 263–268. DOI: 10.1109/ICECCS.2008.24.

[3] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[4] 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.

[5] *ACSL: ANSI ISO C Specification Language*. URL: http://www.frama-c.com/download/acsl.pdf.

[6] 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.

[7] 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.

[8] *Alloy: A Language & Tool for Relational Models*. URL: http://alloy.mit.edu/.

[9] *American Fuzzy Lop*. URL: http://lcamtuf.coredump.cx/afl/.

[10] *Apache Bloodhound*. URL: http://bloodhound.apache.org/.

[11] 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.

[12] *asm.js: an extraordinarily optimizable, low-level subset of JavaScript*. URL: http://www.asmjs.org/.

[13] 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.

[14] Atlassian. *Comparing Workflows*. URL: https://www.atlassian.com/git/tutorials/comparing-workflows (visited on 04/22/2015).

[15] Susan Bell et al. "STAR-Vote: A Secure, Transparent, Auditable, and Reliable Voting System". In: *USENIX EVT/WOTE*. 2013. URL: https://www.usenix.org/conference/evtwote13/workshop-program/presentation/bell.

[16] Cedric Beust and Hani Suleiman. *Next Generation Java Testing*. Addison-Wesley, 2007.

[17] 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. 2009, pp. 19–28. DOI: 10.1109/RE-VOTE.2009.5.

[18] *BitBucket*. URL: http://www.bitbucket.org/.

[19] 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. URL: http://dx.doi.org/10.1007/978-3-319-10082-1_3.

[20] 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.

[21] 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.

[22] 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.

[23] *C0: Specification and Verification in Introductory Computer Science*. URL: http://c0.typesafety.net/.

[24] 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.

[25] Carter Center. *Internet Voting Pilot: Norway’s 2013 Parliamentary Elections*. http://www.cartercenter.org/resources/pdfs/peace/democracy/Carter-Center-Norway-2013-study-mission-report2.pdf. 2014.

[26] *Center for Advanced Software Analysis: Software Tools*. URL: http://casa.au.dk/software-tools/.

[27] Scott Chacon. *Pro Git*. Berkeley, CA New York, NY: Apress,Distributed to the book trade worldwide by Spring Science+Business Media, 2014. ISBN: 978-1484200773.

[28] 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. URL: http://dx.doi.org/10.1007/11555827_8.

[29] David Chaum et al. “Accessible voter-verifiability”. In: *Cryptologia* 33.3 (2009), pp. 283–291.

[30] 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 (2009), pp. 611–627. ISSN: 1556-6013. DOI: 10.1109/TIFS.2009.2034919.

[31] 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.

[32] *Checkstyle*. URL: http://checkstyle.sourceforge.net/.

[33] *CloudFlare, Inc.* URL: http://www.cloudflare.com/.

[34] Alistair Cockburn and Laurie Williams. “The costs and benefits of pair programming”. In: *Extreme programming examined* (2000), pp. 223–247.

[35] *Code Contracts*. URL: http://research.microsoft.com/en-us/projects/contracts/.

[36] *Community Z Tools: Tools for Developing and Reasoning About Z Specifications*. URL: http://czt.sourceforge.net/.

[37] Cas J. F. Cremers. “The Scyther Tool: Verification, Falsification, and Analysis of Security Protocols”. In: *Computer Aided Verification*. Springer, 2008, pp. 414–418.

[38] *CryptoVerif: Cryptographic Protocol Verifier in the Computational Model*. URL: http://cryptoverif.inria.fr/.

[39] *CVK: Crypto Verification Kit*. URL: http://research.microsoft.com/en-us/projects/cvk/.

[40] Alex Delis et al. *Pressing the Button for European Elections 2014: Public attitudes towards Verifiable E-Voting In Greece*. https://drive.google.com/file/d/0B-mtbRwyPn_SdnpMRzBKcEZWUm8/view?usp=sharing. 2014.

[41] Jared DeMott. "Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing". In: *BlackHat and DefCon*. 2007.

[42] David L. Detlefs et al. *SRC Research Report 159: Extended Static Checking*. Compaq Systems Research Center, 1998.

[43] *EasyCrypt: Computer-Aided Cryptographic Proofs*. URL: http://www.easycrypt.info/.

[44] *Eiffel Inspector*. URL: https://docs.eiffel.com/book/eiffelstudio/eiffel-inspector.

[45] *EMMA: A free Java code coverage tool*. URL: http://emma.sourceforge.net/.

[46] Aleks Essex et al. "Punchscan in practice: an E2E election case study". In: *Proceedings of Workshop on Trustworthy Elections*. 2007.

[47] Michael Fagan. "Design and code inspections to reduce errors in program development". In: *Software pioneers*. Springer, 2002, pp. 575–607.

[48] 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, 2013.

[49] *FindBugs*. URL: http://findbugs.sourceforge.net/.

[50] *FogBugz*. URL: http://www.fogcreek.com/fogbugz/.

[51] Python Software Foundation. *doctest — Test interactive Python examples*. 2015. URL: https://docs.python.org/3/library/doctest.html (visited on 05/05/2015).

[52] *Frama-C*. URL: http://www.frama-c.com/.

[53] 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 04/20/2015).

[54] *GitHub*. URL: http://www.github.org/.

[55] 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. URL: http://dx.doi.org/10.1007/978-3-642-32747-6_1.

[56] 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.

[57] *GNATcoverage Coverage Analysis Tool*. URL: http://www.adacore.com/gnatcovarage/.

[58] 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. URL: http://doi.acm.org/10.1145/1375581.1375607.

[59] Rop Gonggrijp et al. "RIES - Rijnland Internet Election System: A Cursory Study of Published Source Code". English. In: *E-Voting and Identity*. Ed. by PeterY.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. URL: http://dx.doi.org/10.1007/978-3-642-04135-8_10.

[60] Trusted Computing Group. *TPM Main Specification*. URL: http://www.trustedcomputinggroup.org/resources/tpm_main_specification.

[61] *How to Vote: Wombat Voting System*. http://www.wombat-voting.com/how-to-vote.

[62] Engelbert Hubbers, Bart Jacobs, and Wolter Pieters. "RIES - Internet Voting in Action". In: *Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International*. Vol. 1. IEEE. 2005, pp. 417–424.

[63] Engelbert Hubbers et al. "Description and analysis of the RIES internet voting system". In: *Report of the Eindhoven Institute for the Protection of Systems and Information*. Faculty of Mathematics and Computer Science Eindhoven University of Technology, 2008.

[64] *HUnit-Plus: A test framework building on HUnit.* URL: https://hackage.haskell.org/package/HUnit-Plus.

[65] Laura Inozemtseva and Reid Holmes. "Coverage is not strongly correlated with test suite effectiveness". In: *Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 435–445.

[66] *JIRA*. URL: https://www.atlassian.com/software/jira.

[67] *JSCert: Certified JavaScript*. URL: http://www.jscert.org/.

[68] *JUnit*. URL: http://www.junit.org/.

[69] 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. URL: http://doi.acm.org/10.1145/357172.357176.

[70] John Launchbury et al. "Application-Scale Secure Multiparty Computation". English. 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. URL: http://dx.doi.org/10.1007/978-3-642-54833-8_2.

[71] Gary T. Leavens et al. *JML Reference Manual*. 2013. URL: http://www.jmlspecs.org/refman/jmlrefman.pdf.

[72] M.V. Linhares et al. "Introducing the modeling and verification process in SysML". In: *IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2007)*. 2007, pp. 344–351. DOI: 10.1109/EFTA.2007.4416788.

[73] Gavin Lowe. "Casper: A compiler for the analysis of security protocols". In: *Computer Security Foundations Workshop*. IEEE. 1997, pp. 18–30.

[74] 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. URL: http://doi.acm.org/10.1145/2499368.2451167.

[75] Neal McBurnett et al. *Scantegrity Responds to Rice Study on Usability of the Scantegrity II Voting Systemesponds to Rice Study on Usability of the Scantegrity II Voting System*. 2014. URL: http://vote.caltech.edu/sites/default/files/Scantegrity%20responds%20to%20Rice%20study%2012-28-14.pdf.

[76] *Méthode B*. URL: http://www.methode-b.com/.

[77] B. Meyer et al. "Programs That Test Themselves". In: *Computer* 42.9 (2009), pp. 46–55. ISSN: 0018-9162. DOI: 10.1109/MC.2009.296.

[78] Bertrand Meyer. *Object-Oriented Software Construction*. 2nd Edition. Prentice-Hall, 1997.

[79] *Mirage OS*. URL: http://www.openmirage.org/.

[80] George C. Necula. "Proof-Carrying Code. Design and Implementation". English. 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. URL: http://dx.doi.org/10.1007/978-94-010-0413-8_8.

[81] *NUnit*. URL: http://www.nunit.org/.

[82] Object Management Group. *UML Human-Usable Textual Notation*. 2004. URL: http://www.omg.org/spec/HUTN/1.0/.

[83] United States. General Accounting Office. *Voters with disabilities: access to polling places and alternative voting methods*. US General Accounting Office, 2001.

[84] *OMG Systems Modeling Language*. URL: http://www.omgsysml.org/.

[85] *OpenCover*. URL: https://github.com/OpenCover/opencover.

[86] *OpenJML*. URL: http://openjml.org.

[87] *OSATE2-Ocarina*. URL: http://www.openaadl.org/osate-ocarina.html.

[88] *OSCE/ODIHR Election Assessment Mission Report*. http://www.osce.org/odihr/elections/netherlands/24322?download=true. 2006.

[89] *Overture Tool: Formal Modeling in VDM*. URL: http://overturetool.org/.

[90] C Pilato. *Version control with Subversion*. Sebastopol, CA: O'Reilly Media, 2008. ISBN: 0596510330.

[91] *PMD*. URL: http://pmd.sourceforge.net/.

[92] 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. URL: http://doi.acm.org/10.1145/2043556.2043566.

[93] 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.

[94] 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. URL: http://dx.doi.org/10.1007/978-3-642-12980-3_15.

[95] *QuickCheck: Automatic testing of Haskell programs*. URL: https://hackage.haskell.org/package/QuickCheck.

[96] *RAISE—Rigorous Approach to Industrial Software Engineering*. URL: http://spd-web.terma.com/Projects/RAISE/.

[97] *Redmine*. URL: http://www.redmine.org/.

[98] *ReSharper*. URL: https://www.jetbrains.com/resharper/.

[99] Noel Runyan. "Improving access to voting: A report on the technology for accessible voting systems". In: *Retrieved October* 1 (2007), p. 2008.

[100] P.Y.A. Ryan et al. "Prêt à Voter: a Voter-Verifiable Voting System". In: *Information Forensics and Security, IEEE Transactions on* 4.4 (2009), pp. 662–673. ISSN: 1556-6013. DOI: 10.1109/TIFS.2009.2033233.

[101] *SAW: The Software Analysis Workbench*. URL: http://saw.galois.com/.

[102] *Security Review: Helios Online Voting*. https://cubist.cs.washington.edu/Security/2009/03/13/security-review-helios-online-voting/. 2009.

[103] N. Shankar. "PVS: Combining specification, proof checking, and model checking". English. 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. URL: http://dx.doi.org/10.1007/BFb0031813.

[104] Claire M. Smith. *Convenience Voting and Technology: The Case of Military and Overseas Voters (Elections, Voting, Technology)*. Palgrave Macmillan, 2014. ISBN: 1137398582. URL: http://www.amazon.com/Convenience-Voting-Technology-Military-Elections/dp/1137398582\%3FSubscriptionId\%3D0JYN1NVW651KCA56C102\%26tag\%3Dtechkie-20\%26linkCode\%3Dxm2\%26camp\%3D2025\%26creative\%3D165953\%26creativeASIN\%3D1137398582.

[105] Claire M Smith. "Time to Move: Overseas and Military Voter State Policy Innovation". In: (2011).

[106] *SourceForge*. URL: http://www.sourceforge.net/.

[107] *SPARK 2014*. URL: http://www.spark-2014.org/.

[108] *SPARK Pro*. URL: http://www.adacore.com/sparkpro/.

[109] *StyleCop*. URL: http://www.stylecop.com/.

[110] *SUnit: The mother of all unit testing frameworks*. URL: http://sunit.sourceforge.net/.

[111] 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.

[112] *The Coq Proof Assistant*. URL: https://coq.inria.fr/.

[113] "The Daikon system for dynamic detection of likely invariants". In: *Science of Computer Programming* 69.1–3 (2007), pp. 35–45. DOI: http://dx.doi.org/10.1016/j.scico.2007.01.015".

[114] *The Haskell Lightweight Virtual Machine (HaLVM)*. URL: https://galois.com/project/halvm/.

[115] 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. URL: http://doi.acm.org/10.1145/358198.358210.

[116] 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, 2008, pp. 134–153. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=81193.

[117] James E Tomayko. "A comparison of pair programming to inspections for software defect reduction". In: *Computer Science Education* 12.3 (2002), pp. 213–222.

[118] *Trac*. URL: http://trac.edgewall.org/.

[119] Georgios Tsoukalas et al. "From Helios to Zeus". In: *USENIX EVT/WOTE*. 2013. URL: https://www.usenix.org/conference/evtwote13/workshop-program/presentation/Tsoukalas.

[120] *UPPAAL*. URL: http://www.uppaal.org/.

[121] U.S. Election Assistance Commission. *UOCAVA Pilot Program Testing Requirements—August 25, 2010*. 2010. URL: http://www.eac.gov/assets/1/Documents/UOCAVA_Pilot_Program_Testing\%20Requirements\%20August\%208\%202010.pdf.

[122] *Verified Software Toolchain*. URL: http://vst.cs.princeton.edu/.

[123] *Verifying Multi-threaded Software with Spin*. URL: http://spinroot.com/.

[124] Kim Walden. *Seamless object-oriented software architecture : analysis and design of reliable systems*. New York: Prentice Hall, 1995. ISBN: 0130313033.

[125] David A. Wheeler. "Fully Countering Trusting Trust through Diverse Double Compilation". PhD thesis. George Mason University, 2009. URL: http://www.dwheeler.com/trusting-trust/.

[126] *Xen Project*. URL: http://www.xenproject.org/.

[127] Xuejun Yang et al. "Finding and understanding bugs in C compilers". In: *ACM SIGPLAN Notices*. Vol. 46. 6. ACM. 2011, pp. 283–294.

[128] 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.

[129] *YouTrack*. URL: https://www.jetbrains.com/youtrack/.

[130] 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. URL: http://dx.doi.org/10.1007/978-3-642-38980-1_28.

[131] Daniel M. Zimmerman and Rinkesh Nagmoti. "JMLUnit: The Next Generation". In: *International Conference on Formal Verification of Object-Oriented Software (FoVeOOS 2010)*. Paris, France, 2010.

[132] Philip Zimmermann. *The Official PGP User's Guide*. Cambridge, MA: The MIT Press, 1995. ISBN: 0-262-74017-6.