

RICE UNIVERSITY

VoteBox: A tamper-evident, verifiable voting machine

by

Daniel Robert Sandler

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Dan S. Wallach (Chair)
Associate Professor of Computer Science

T. S. Eugene Ng
Assistant Professor of Computer Science

Michael D. Byrne
Associate Professor of Psychology

HOUSTON, TEXAS

APRIL 2009

ABSTRACT

VoteBox: A tamper-evident, verifiable voting machine

by

Daniel Robert Sandler

This thesis details the design and implementation of VOTEBOX, a new software platform for building and evaluating secure and reliable electronic voting machines. Current electronic voting systems have experienced many high-profile software, hardware, and usability failures in real elections. Recent research has revealed systemic flaws in commercial voting systems that can cause malfunctions, lose votes, and possibly even allow outsiders to influence the outcome of a national election. These failures and flaws cast doubt on the accuracy of elections conducted with electronic systems and threaten to undermine public trust in the electoral system.

While some consequently argue for total abandonment of electronic voting, VOTEBOX shows how a combination of security, distributed systems, and cryptographic principles can yield trustworthy and usable voting systems. It employs a pre-rendered user interface to reduce the size of the runtime system that must be absolutely trusted. VOTEBOX machines keep secure logs of essential election events, allowing credible audits during or after the election; they are connected using the Auditorium, a novel peer-to-peer network that replicates and intertwines secure logs in order to survive failure, attack, and poll worker error. While the election is ongoing, any voter may choose to challenge a VOTEBOX to immediately produce cryptographic proof that it will correctly and faithfully cast ballots.

This work uniquely demonstrates how these disparate approaches can be used in concert to increase assurance in a voting system; the resulting design also offers a number of pragmatic benefits that can help reduce the frequency and impact of poll worker or voter errors. VOTEBOX is a model for new implementations, but its component techniques can be practically applied to existing systems. VOTEBOX ideas should therefore find their way into commercial electronic voting machines as well as other problem domains in which tamper-evidence, robustness, and verifiability are crucial.

ACKNOWLEDGEMENTS

This document is the product of advice, ideas, code, and conversation of many people in addition to my own. I mean this in a mathematical sense: their help has *multiplied* my own contribution, resulting in a body of work of which I am very proud.

Dan Wallach, my thesis adviser, took me in when I was without a country in my department. He saw in me a potential security researcher, voting expert, and computer scientist; he gave me a research field and the freedom to find and solve the problems that would satisfy my own intellectual curiosity. Eugene Ng and Mike Byrne also served on my thesis committee, and I thank them for their tireless push to make this work more rigorous and more bold.

Kyle Derr is my co-author several times over; with him I designed most of the guts of VOTEBOX, and the software bears his fingerprints. Working with Kyle was like having my own graduate student.¹ Scott Crosby and Ted Torous also made substantial and essential contributions to the published work about VOTEBOX, for which I am grateful. Emily Fortuna, George Mastrogiannis, Kevin Montrose, and Corey Shaw each made their own mark on the VOTEBOX codebase, and I appreciate the opportunity to have worked with each of them.

Many scholars have assisted with VOTEBOX in some way. Kristen Greene and Sarah Everett helped us design the VOTEBOX user experience, while we helped them conduct their experiments; I am proud of the collaboration between the Computer Science and Psychology departments and hope that such interactions continue after I am gone. Ben Adida and Brent Waters have served as my cryptographic conscience during this project, and I am thankful for their suggestions (and mathematics). Avi Rubin has contributed to this work with his comments and suggestions, but more importantly by helping to make possible the multi-institutional ACCURATE center, through which the NSF has

¹I mean this as a compliment.

funded me over the course of this project.²

Tracy Volz and Jan Hewitt are directly responsible for the marked improvement in my oral and written communication that occurred during my graduate career, and I am grateful for the confidence such improvement brings. Seth Nielson has been a friend and colleague throughout my time in the Computer Security Lab, and I have appreciated his help as we have navigated graduate school together. The administrators of the Computer Science department, in particular Darnell Price, have been an indispensable part of my graduate career as well, and I will miss their unfailing assistance. The job of a sysadmin is a difficult and venerated one; Michael Lightfoot has expertly supported our lab's eccentric computing needs. I owe a tremendous debt of thanks to Luay Nakhleh and Keith Cooper³ for their advice and confident support over the last five years.

Finally, I thank my family, which increased in number during the production of this thesis. For my whole life, my parents and my brother⁴ have always encouraged and supported me in my education and my work; my parents- and siblings-in-law, somewhat later to the game, are no less dedicated to my success. I know I can count on their trust and confidence now and in the future. My wife Erin is an essential part of my achievement, and I hers; I have every expectation that we will continue to inspire and support one another through our careers and lives together. And to my son, Nathan: Thank you for allowing me to leave you behind when you were just two weeks old so that I could present this work at a conference [86]. By the time you read this you will have figured out that I enjoy starting things more than I enjoy finishing them, and I want you to know that I finished this most of all for you.

²National Science Foundation grant CNS-0524211.

³To whose office door I suspect I have worn a path at this point.

⁴For his entire life, anyway.

CONTENTS

1	Introduction	1
2	Background	4
2.1	The move to electronic voting in the United States	4
2.2	Criticism of e-voting	5
2.3	Advantages of computerized voting	7
2.3.1	Accessibility	8
2.3.2	Flexibility	8
2.3.3	Usability	9
2.4	Toward software independence	9
2.5	Reducing the trusted computing base	11
2.6	The importance of audit logs	12
2.7	Testing	13
2.8	Cryptography and e-voting	15
2.9	“Paper cryptography”	17
2.10	Paper plus computers: Optical scan and VVPAT	19
2.11	Internet voting	20
2.12	Byzantine faults in distributed systems	22
3	Auditorium	23
3.1	Introduction	23
3.2	March, 2006: Laredo, Webb County, Texas	24
3.2.1	Findings	25
3.3	The design of Auditorium	29
3.3.1	Requirements for auditable voting systems	29
3.3.2	How I learned to stop worrying and love the network	30
3.3.3	Secure logging	30
3.3.4	Entangled timelines	31
3.3.5	Auditorium: n-way entanglement and replication	32
3.3.6	The Auditorium broadcast network	33
3.3.7	Voting in the Auditorium	35
3.4	Robustness to failure and attack	40
3.4.1	Failures and mistakes (and attacks masquerading as such)	40
3.4.2	Mega attacks	42
3.4.3	Software tampering	43

4	Verifiability: the cast-as-intended challenge	45
4.1	Introduction	45
4.2	Voter privacy; encryption	46
4.3	Tallying encrypted ballots	47
4.3.1	Ballots as counter vectors	48
4.3.2	Homomorphic encryption of counters	48
4.4	Verifiability through ballot challenge	49
4.4.1	Immediate ballot challenge	49
4.4.2	Implications of the challenge scheme	52
4.5	Procedures: administering a VOTEBox election	52
4.5.1	Before the election	53
4.5.2	Opening the polls	53
4.5.3	Casting votes	53
4.5.4	Closing the polls	54
4.6	Security of the cryptosystem and ballot challenge technique	55
4.6.1	Ballot decryption key material	55
4.6.2	Covert channels in randomized ciphers	55
4.7	Attacks on the challenge system	55
4.8	Verifying the tabulation	58
5	Querifier: Secure log analysis	60
5.1	Integrity checking in secure logs	60
5.1.1	Application-specific properties	60
5.2	A language for log properties	62
5.2.1	Domain of expression	62
5.2.2	Relations	63
5.2.3	Logical expressivity	64
5.3	Algorithms used in Querifier	64
5.3.1	Evaluation of tuple logic expressions	64
5.3.2	Algorithms for ordering log entries	66
5.3.3	Incremental verification	70
5.4	Implementation	73
5.4.1	Introduction	73
5.4.2	Operation	73
5.4.3	Experimental setup	75
5.4.4	Results	77
6	Properties of the user interface	82
6.1	Pre-rendering for assurance	82
6.2	Ballot creation	84
6.3	Human factors experimentation	86
7	The VoteBox prototype	88
7.1	Introduction	88
7.2	Software implementation notes	88
7.2.1	Secure software design	88

7.2.2	Insecure software design	91
7.2.3	Concrete representation of data	92
7.3	Metrics	92
7.3.1	Code size	92
7.4	Performance	93
7.4.1	Network load	94
7.4.2	Bandwidth requirements of the challenge scheme	95
7.4.3	Storage	97
7.4.4	CPU demands of encryption	98
8	Extension: remote voting	101
8.1	Introduction	101
8.2	Provisional and postal voting	102
8.3	Security and privacy of remote voting	104
8.3.1	Conventional approaches	104
8.3.2	Goals for a networked replacement	104
8.4	RemoteBox: connecting remote precincts over the net	106
8.4.1	Remote electronic voting	106
8.4.2	Ballot definitions	107
8.4.3	Cryptographic and pragmatic details	107
8.5	Summary	109
9	Conclusion	110
A	Auditorium protocol for voting	113
A.1	Auditorium messages	113
A.1.1	Data structures	113
A.1.2	Messages	115
A.2	Voting messages	117
A.2.1	Sent by supervisor	118
A.2.2	Sent by booths	121
B	Highlight graphics	124
C	Internet resources	126

LIST OF FIGURES

3.1	Impounded voting machines in Laredo, TX.	25
3.2	An iVotronic event log.	26
3.3	Machine showing the wrong date.	28
3.4	Likely test votes.	28
3.5	Flow of time in the Auditorium.	32
3.6	Data survival under various replication factors.	35
3.7	Configuration of machines in a polling place.	38
4.1	Challenge flow chart.	50
4.2	Voting with ballot challenges.	51
4.3	Final VOTEBOX screen: challenge, or record?	57
5.1	Querifier components and operation.	73
5.2	S-expression representation of logical rules.	74
5.3	Incremental evaluation.	78
5.4	Graph search.	79
5.5	Summary of implementations.	80
5.6	Ruleset comparison.	81
6.1	Sample VOTEBOX page.	83
6.2	The VOTEBOX ballot creator.	85
8.1	Voting with RemoteBox.	105

LIST OF TABLES

5.1	Summary of graph-of-time search algorithms.	67
7.1	Size of the VOTeBOX trusted codebase.	93
7.2	Bandwidth of Auditorium messages involved in a voting session.	96

CHAPTER 1

INTRODUCTION

Electronic voting is at a crossroads. Having been aggressively deployed across the United States as a response to flawed paper and punch-card voting in the 2000 U.S. national election, digital-recording electronic (DRE) voting systems are themselves now seen as flawed and unreliable. They have been observed in practice to produce anomalies that cannot be adequately explained by the poll workers, voters, or manufacturers. In some cases, voters have observed “vote flipping” phenomena in which a user action to choose one candidate has caused another to become selected instead. In others, elections officials have observed irregular voting patterns, such as anomalously high undervoting (abstention in one or more races) in some districts and not others. Frequently, audit logs kept by the voting machines prove inconclusive or even incomplete, offering little assistance to those attempting to assemble a clear picture of the events transpiring on election day.

As DRE systems proliferated, computer scientists and other scholars began to investigate the possible hazards of using digital systems to capture votes, and in particular to analyze the specific e-voting systems in use in the U.S. This line of research culminated in 2007 with independent security reviews commissioned by the states of California and Ohio; these landmark investigations represent the first time that outside experts and scholarly researchers have been given access to the inner workings of these machines, including source code and development documentation. The results of the reviews are universally damning: they reveal that every DRE voting system currently in widespread use in the United States has severe deficiencies in design and implementation. While they were most likely developed with every intention of safeguarding the vote, these systems have demonstrated through failure and under scrutiny that they cannot and should not be trusted with the vote. (Chapter 2

describes these research efforts and analyses in more detail.)

The consequence of these experiences and examinations, as of 2009, is widespread backlash against DRE systems. Many jurisdictions are now decertifying or restricting the use of electronic voting systems. However, by abandoning the very *idea* of electronic voting we also sacrifice several advantages offered by these systems to voters and administrators.

Accessibility. Many voters are physically unable to vote on a paper ballot; electronic voting systems help these voters cast ballots without the assistance of another person in the voting booth. Voters with vision impairments can don headphones and interact with an electronic voting system outfitted with an audio interface, for example.

Flexibility. Unencumbered by the physical limitations of a finite and static paper ballot, e-voting systems can support potentially many ballot designs of arbitrary length.

Usability. Computerized voting systems have the potential for much richer user experiences than paper, including feedback that may be able to help voters avoid mistakes. Anecdotal evidence, as well as recent human factors studies (using VOTEBOX), show that voters have a strong subjective preference for electronic voting as well.

The serious security and reliability problems of *current* DRE voting systems must not deter us from striving to develop systems with these valuable properties. This, then, is the purpose of the VOTEBOX project: to explore techniques for building *trustworthy* DRE-style voting systems. In this thesis I detail the design of our VOTEBOX system, which achieves this end by composing research techniques in the areas of security, distributed systems, and cryptography. In particular, VOTEBOX demonstrates the following key techniques:

Auditorium (Chapter 3), a novel hybrid of secure logging and peer-to-peer networking that connects VOTEBOXes in a polling place and provides *resistance to data loss*, even in extreme situations, and *tamper evidence* in those situations where data loss cannot be prevented. The logs produced by Auditorium contain enduring proofs of the correct operation of the system on election day. Correctness, however, can be a complex predicate, involving the existence and

order of multiple critical events in the record; rather than solve the log analysis problem for the voting domain only, I developed Querifier (Chapter 5), a general-purpose tool that can apply any set of logical rules to a broad class of secure logs.

Verifiability (Chapter 4), in the form of a ballot challenge system. Any ballot may potentially serve as an election-day test of the system, offering cryptographic proof that the voter’s intended choices have been correctly captured for tallying, regardless of any defects in the underlying software (“software independence”). This approach, which offers voters a way to initiate real-time audits of voting equipment, is based on work by Benaloh [11]; VOTEBOX contributes an adaptation of this work to the Auditorium environment that makes this cryptographic technique usable in practice.

Reduced codebase (Chapter 6), an important consideration when attempting to analyze, certify, or audit a voting system. While the above techniques help detect and recover from problems, preventing them in the first place involves reading and understanding the source code, causing the size and simplicity of that code to become a concern. We directly apply the pre-rendered user interfaces (PRUI) technique, first proposed for e-voting assurance by Yee [105], to evict complexity from the VOTEBOX software that must operate correctly on election day.

VOTEBOX thus contributes novel inventions (Auditorium), improvements of state-of-the-art work (ballot challenge), and fresh implementations of best practices (PRUI). Moreover, it uniquely demonstrates how disparate results from computer science and e-voting research may be combined to magnify the effects of each. The fusion of these approaches yields additional pragmatic benefits, including an improvement to the conventional process of postal voting (Chapter 8). The VOTEBOX system has been designed to be robust to accidental failures that commonly occur in elections as well as the theoretical dangers inveighed against by the research community; we can therefore trust a VOTEBOX—or any electronic voting system following its design blueprint—with the vote.

CHAPTER 2

BACKGROUND

2.1 The move to electronic voting in the United States

The conclusion of the 2000 U.S. Presidential election illustrated starkly the limitations of much of the voting technology in widespread use across the country at that time. Punch cards (used by about a third of voters in 2000 [6]) fared even worse: Mechanical problems, such as incompletely-punched cards (a condition that came to be known as “hanging chads”), created cast ballots that were ambiguous. Usability problems, particularly with “butterfly ballots” that do not clearly indicate which candidate a particular punch location may correspond to, caused ballots to be unambiguously cast for the wrong person.

Lever-based voting machines ($\frac{1}{6}$ of voters [6]) had their own problems. Based on technology originally invented more than a century earlier [58] (and out of manufacture since 1982 [57]), they are cumbersome, costly to maintain, and prone to mechanical failures. More importantly, they offer no auditability; because they operate by incrementing physical counters, a voter’s choices are not captured in a single recountable ballot but instead added immediately and irrevocably to the tally. The Caltech/MIT Voting Project [6] estimated that as many as 2 million votes were lost in 2000 due to these sorts of technical problems; its recommendation in 2001 was that outdated voting technology be replaced and that federal money be set aside for this purpose.

To this report and others like it in the wake of this debacle, the United States Congress responded by passing the Help America Vote Act of 2002 (HAVA) [3], Title I of which allocated money for states to improve or replace aging voting technologies:

SEC. 102. REPLACEMENT OF PUNCH CARD OR LEVER VOTING MACHINES.**(a) ESTABLISHMENT OF PROGRAM.—**

(1) **IN GENERAL.**—Not later than 45 days after the date of the enactment of this Act, the Administrator shall establish a program under which the Administrator shall make a payment to each State eligible under subsection (b) in which a precinct within that State used a punch card voting system or a lever voting system to administer the regularly scheduled general election for Federal office held in November 2000 (in this section referred to as a “qualifying precinct”).

(2) **USE OF FUNDS.**—A State shall use the funds provided under a payment under this section (either directly or as reimbursement, including as reimbursement for costs incurred on or after January 1, 2001, under multi-year contracts) to replace punch card voting systems or lever voting systems (as the case may be) in qualifying precincts within that State with a voting system (by purchase, lease, or such other arrangement as may be appropriate) that—

- (A) does not use punch cards or levers;
- (B) is not inconsistent with the requirements of the laws described in section 906; and
- (C) meets the requirements of section 301.

[3, Title I, Sec. 102 (a).]

Spurred by this mandate, many states did indeed choose to replace their voting systems. In 2004, the deadline established in HAVA for states to participate in the equipment replacement program, 32% of registered voters were in precincts that had switched to optical-scan systems, and 29% of voters would vote on DRE systems. [32]

2.2 Criticism of e-voting

In February 2003, activist Bev Harris of Black Box Voting¹ disclosed [46] that she had found, via Web searching, a public ftp server that allowed access to source code from Diebold Election Systems, Inc. (DESI),² manufacturers of the widely-deployed AccuVote touch-screen DRE system. She subsequently made the Diebold files (since removed from the ftp server) available and identified several ways in which attackers might be able to alter votes and cover their tracks by tampering with audit logs. [45]

¹<http://www.blackboxvoting.org>

²As of August 2007, Diebold’s e-voting subsidiary is now known as “Premier Election Solutions.”

Computer scientists and security researchers, long concerned about the possible dangers of electronic voting but stymied by the secrecy surrounding commercial e-voting systems, promptly examined the Diebold source code for flaws. Doug Jones³ posted an early analysis of his findings among the Diebold files [59]. That same month, another group of computer scientists posted a detailed technical report (the so-called “Hopkins paper” was later published as Kohno et al. [66]) analyzing the source code to the DRE, identifying a wide variety of flaws, none of which require source code access to perpetrate. These included:

- A weak voter-authentication protocol allows anyone with a specially-designed smartcard to cast an unlimited number of votes.
- Ballots and logs are encrypted, but the same symmetric DES key⁴ is used on every AccuVote (and had been in use since 1998).
- Using features of the system designed to facilitate software upgrades in the field, an attacker can introduce malicious code into the system.

Other studies followed [50, 51, 76, 93, 52, 28, 36], including a Princeton study [35] in which researchers found that anyone with physical access to a Diebold voting machine could introduce a “voting machine virus” into the system. The regular process of collecting votes at the end of an election requires poll workers to use flash memory cards to download ballot data from each; the same memory card can be (and is usually) used to download votes from many machines in turn. In combination with the malicious-software-upgrade flaw identified by the Hopkins paper, an attacker can introduce (to a “patient zero” machine) carefully-crafted code that will *spread* from machine to machine as votes are collected.

In 2007, responding both to the alarms sounded by researchers and the accumulating news reports of e-voting failures, the states of California and Ohio each commissioned comprehensive reviews of voting systems used in those states. Debra Bowen, California Secretary of State, arranged for

³Dr. Jones has a long history of interest in psephology (and voting equipment in particular) predating the Diebold code leak and even the problems in 2000; he explains that he saw an early version of the AccuVote when it was called the I-Mark Electronic Ballot Station [59].

⁴`#define DESKEY ((des_key*)"F2654hD4")`

a “top-to-bottom review” in which source code and documentation for voting systems from Hart InterCivic, Diebold, and Sequoia were acquired by the state and submitted to teams of security experts for study. Other teams behaved as “red teams,” instructed to attack the physical voting machines. A similar arrangement was created by Secretary of State Jennifer Brunner for Ohio’s Evaluation & Validation of Election-Related Equipment, Standards & Testing (EVEREST) project.

The results were universally damning [53, 18, 13, 75, 17]. Serious security problems were discovered in every voting system certified for use in California and Ohio, including DRES and optical scanners. As a result, California promptly limited the use of DRES and require manual audits of paper ballots to double-check electronic tallies. Brunner recommended that Ohio move to centrally-tabulated optical-scan ballots and expand absentee and early voting.

While both security flaws and software bugs have received significant attention, a related issue has also appeared numerous times in real elections using DRES: operational errors and mistakes. Chapter 3 recounts my experience investigating anomalous electronic voting records in a 2006 primary election in Webb County, Texas. More recently, in the January, 2008 Republican presidential primary in South Carolina, several ES&S iVotronic systems were incorrectly configured subsequent to pre-election testing, resulting in those machines being inoperable during the actual election. “Emergency” paper ballots ran out in many precincts and some voters were told to come back later [26].

2.3 Advantages of computerized voting

The wave of scholarly criticism and public malfunctions has fed a broadening public backlash against electronic voting. Today’s commercial DRE voting systems are fragile, insecure, and opaque; they are therefore highly deserving of such condemnation. In general, software systems cannot be trusted to be entirely free of bugs, nor can poll workers and election officials be expected to flawlessly operate complex voting equipment (electronic or otherwise).

Many critics have proposed, and jurisdictions have responded by, returning to some form of paper-based voting, including *optical scan* (also called *mark sense*) systems. Voters using such a system mark a paper ballot in such a way that it can be scanned by a computer, inspired by similar technology developed for standardized testing. [57]

A piece of paper can be verified correct by the voter before personally placing it in a ballot box. Recounts may be performed of these physical records, which are assumed to be harder to forge and easier to authenticate than electronic data. By rushing to return to paper ballots, however, we necessarily abandon the many benefits (introduced in Chapter 1) that fully electronic voting technology affords.

2.3.1 Accessibility

Even in jurisdictions where DRES are being phased out, precincts are commonly required to have one or two electronic voting machines on-hand expressly for purposes of accessibility. DRE systems commonly support audio feedback (in the form of spoken prompts through attached headphones) that makes voting possible for users with low or no vision. Voters with disabilities that prevent them from marking paper can use so-called “sip and puff” assistive devices to provide input to DRES as well. The ability to cast a ballot without assistance is an essential part of the secret vote, and so for this reason alone DRES are a required component of future elections.

2.3.2 Flexibility

A paper ballot is finite; it can only hold so many races, so many names, without becoming illegible. In a race such as the 135-candidate 2003 California gubernatorial recall election, [7], the ballot design flexibility afforded by electronic voting is not a trivial concern. Beyond merely being unbounded in length, ballots presented on a computer screen can change font size, color, or other presentation details to accommodate the vision or preference of the voter. They may allow the voter to switch back and forth between languages (whereas including two or even three languages on a paper ballot further reduces the amount of information that can be printed on it).

Flexibility is an even greater asset for elections administrators. Electronic systems obviate the need to physically transport large boxes of paper ballots to the polling place on election day and back again for tallying (a process which is itself greatly accelerated in the digital domain). An unlimited number of ballot designs can be instantly deployed in any quantity in any polling place, which is an enormous boon for early voting (in which voters from many home precincts converge on a single

polling place). This “weightless” property of e-voting systems is a crucial enabling factor for *election day vote centers*, a trend designed to dramatically improve the convenience (and, it is argued, the turnout) of voting. Pioneered in Colorado, vote centers replace many small residential polling places in a jurisdiction (e.g. a county) with a few large voting facilities located in population centers, not unlike early voting (albeit with many more voters). [96] Such an arrangement must accommodate tens or hundreds of thousands of votes cast on hundreds of ballot styles assigned to each voter based on his residence address, an infeasible proposition with paper ballots but quite tractable with electronic systems.

2.3.3 Usability

DRES are general-purpose computers, and as such they have the potential to offer user experiences that are every bit as interactive, helpful, and fun as any personal computer or smartphone. Today’s electronic voting systems already provide valuable feedback that can avoid voter mistakes that would cause a paper ballot to be disqualified; by enforcing the correct maximum number of votes in a race, the DRE effectively prevents the user from overvoting. DRES can also alert voters to unintentional undervotes by offering a final *review screen* that allows a voter to ensure she has cast votes in each contest of interest.

Recent user studies, conducted by human factors researchers at Rice using VOTEBOX, reveal that electronic voting has a very high subjective usability when compared with other conventional voting schemes: paper ballots, punch cards, and lever machines. [33] This corroborates anecdotal accounts that voters do in fact prefer to vote on digital systems, and may indicate that voters may be opposed to efforts to replace DRES with more conventional voting systems.

2.4 Toward software independence

Recently, the notion of *software independence* has been put forth by Rivest and other researchers seeking a way out of this morass:

A voting system is software-independent if an undetected change or error in its software cannot cause an undetectable change or error in an election outcome. [80]

Such a system produces results that are verifiably correct or incorrect irrespective of the system's implementation details; any software error, whether malicious or benign, cannot yield an erroneous output masquerading as a legitimate cast ballot. In general, the voter's intent is always represented accurately and faithfully in the final tally. This idea generalizes beyond software, of course; a *mechanism independent* voting system is impervious to problems of implementation irrespective of technology. We might say that the punch-card voting systems used in Florida in 2000 (as described earlier in this chapter) were not mechanism independent, because an undetected error in the system (in this case, a failure to punch cards completely) was able to cause an undetectable error in the outcome (the voter's original intent was lost on certain cards).

When verifying that a voting system is software- (or mechanism-) independent, one typically breaks this problem down into two pieces, each of which must be proven:

Cast as intended. Each vote must be *cast*—permanently recorded in some way—as an accurate and unambiguous representation of the voter's *intent*.

Counted as cast. Having been *cast* correctly, the votes must now be *counted* accurately.

To satisfy the *counted as cast* test, a voting system must be able to prove that the output tally matches exactly the input cast ballots. When those ballots are plain text (i.e. not encrypted), this can be achieved by publishing both the set of ballots and the computed total, but ballot encryption complicates matters. In the non-electronic realm, this problem is addressed by recounts, sometimes conducted by hand, but without additional measures a recount of an electronic system is meaningless because of the *cast as intended* problem.

The *cast as intended* property is the harder of the two to satisfy, particularly for a DRE, whose electronic workings are difficult to reveal. A voter may directly examine the piece of paper she is about to place in a physical ballot box, but she may not directly examine the bits stored on her behalf on a flash card deep inside a DRE (and even if she could, those bits might be encrypted). Nonetheless,

VOTEBOX includes measures that allow voters to verify this property, as well as the *counted as cast* property. The specific techniques involved are described in Chapter 4.

2.5 Reducing the trusted computing base

Software independence allows voters and administrators to verify the correct operation of a voting machine. What if that verification fails, due to a bug or design flaw or even malice? It is crucial to *detect* that problems exist, but to prevent them from occurring in the first place, voting software must be carefully audited well before election day.

This is a laborious process, involving human auditors. One approach to mitigating the difficulty of this task is to draw a line around the set of functions that are essential to the correctness of the vote and aggressively evict complexity from that set. If assurance can come from reviewing and auditing voting software, then it should be easier to review and ultimately gain confidence in a smaller software stack.

Pre-rendered user interface (PRUI) is an approach to reducing the amount of voting software that must be reviewed and trusted [105]. Exemplified by Pvote [104], a PRUI system consists of a ballot definition and a software system to present that ballot. The ballot definition comprises a state machine and a set of static bitmap images corresponding to those states; it represents what the voter will see and interact with. The software used in the voting machine acts as a virtual machine for this ballot “program.” It transitions between states and sends bitmaps to the display device based on the voter’s input (e.g., touchscreen or keypad). The voting VM is no longer responsible for text rendering or layout of user interface elements; these tasks are accomplished long in advance of election day when the ballot is defined by election officials.

A ballot definition of this sort can be audited for correctness independently of the voting machine software *or* the ballot preparation software. Even auditors without knowledge of a programming language can follow the state transitions and proofread the ballot text (already rendered into pixels). The voting machine VM should still be examined by software experts, but this code—critical to capturing the user’s intent—is reduced in size and therefore easier to audit. Pvote comprises just 460 lines of Python code, which (even including the Python interpreter and graphics libraries) compares favor-

ably against current DRES: the AccuVote TS involves over 31,000 lines of C++ running atop Windows CE [104]. Chapter 6 shows how VOTEBOX applies the PRUI technique to reduce its own code footprint.

Compartmentalization, an alternative approach to reducing the trusted computing base, is demonstrated by Sastry et al. [90]. The trusted program modules in their system are forced to be small and clearly separated by dedicating a separate computer to each. The modules operate on isolated CPUs and memory, and are connected with wires that may be observed directly; each module may therefore be analyzed and audited independently without concern that they may collude using side channels, although there is still a possibility of leakage through other means (storing vote data or transmitting it via radio, for example). Additionally, the modules may be powered off and on between voters to eliminate the possibility of state leaking from voter to voter due to a bug. (VOTEBOX incorporates some of this insight into its design, as shown in Chapter 7.)

2.6 The importance of audit logs

Even trustworthy systems can be misused, and this problem occurs with unfortunate regularity in the context of voting. In the case of electronic voting, poll workers are expected to correctly deploy, operate, and maintain large installations of unfamiliar computer systems. DRE vendors offer training and assistance, but on election day there is typically very little time to wait for technical support while voters queue up.

In fact, operational and procedural errors can (and do) occur during elections. Machines unexpectedly lose power, paper records are misplaced, hardware clocks are set wrong, and test votes (see Section 2.7 below) are mingled with real ballots. Sufficient trauma to an honest DRE may result in the loss of its stored votes.

In the event of an audit or recount, comprehensive records of the events of election day are essential to establishing (or eroding) confidence in the results despite these kinds of election-day mishaps. Many DRES keep electronic audit logs, tracking election day events such as “the polls were opened” and “a ballot was cast,” that would ideally provide this sort of evidence to *post facto* auditing efforts. Unfortunately, current DRES entrust each machine with its own audit logs, making them no safer from failure or accidental erasure than the votes themselves. Similarly, the audit logs kept by cur-

rent DRES offer no integrity safeguards and are entirely vulnerable to attack; any malicious party with access to the voting machine can trivially alter the log data to cover up any misdeeds.

The Auditorium system (Chapter 3) confronts this problem by using techniques from distributed systems and secure logging to make audit logs into believable records. All voting machines in a polling place are connected in a private broadcast network; every election event that would conventionally be written to a private log is also “announced” to every voting machine on the network, each of which *also* logs the event. Each event is bound to its originator by a digital signature, and to earlier events from other machines via a *hash chain*. The aggressive replication protects against data loss and localized tampering; when combined with hash chains, the result is a hash mesh [92] encompassing every event in the polling place. An attacker (or an accident) must now successfully compromise every voting machine in the polling place in order to escape detection.

2.7 Testing

Regrettably, the conventional means by which today’s commercial voting machines are deemed trustworthy is through testing. Long before election day, the certification process typically includes some amount of analysis and testing by independent testing authorities. The Election Assistance Commission, created by HAVA in 2002, is responsible for promulgating recommendations for evaluating voting systems, including a set of guidelines for testing [99, Volume II].⁵

The tests recommended by the EAC, including examination of source code, may be performed by one of four (as of this writing [31]) EAC-certified testing laboratories. This same openness is not afforded to other experts and scholars in the field; in particular, the source code of voting systems has traditionally been considered a trade secret by vendors and is not disclosed to researchers. Therefore, there has long been concern that the necessary expertise required to identify weaknesses and problems is not being brought to bear on the problem. [56] As proof we need look no further than the litany of problems (highlighted in Section 2.2) identified by outside researchers *without* the assistance of voting system vendors.

Complementary to certification testing, *logic and accuracy* (L&A) testing is a common black-box

⁵These guidelines are voluntary; no vendor or jurisdiction is bound by them.

testing technique practiced by elections officials, typically in advance of each election. L&A testing typically takes the form of a mock election: a number of votes are cast for different candidates, and the results are tabulated and compared against expected values. The goal is to increase confidence in the predictable, correct functioning of the voting systems on election day.

Similar to L&A testing, *parallel* tests are performed on election day with a small subset of voting machines selected at random from the pool of “live” voting systems. The units under test are sequestered from the others; as with L&A testing, realistic votes are cast and tallied. By performing these tests on election day with machines that would otherwise have gone into service, parallel testing is assumed to provide a more accurate picture of the behavior of other voting machines at the same time.

The fundamental problem with these tests is that they are artificial: the conditions under which the test is performed are not identical to those of a real voter in a real election. It is reasonable to assume that a malicious piece of voting software may look for clues indicating a testing situation (wrong day; too few voters; evenly-spread voter choices) and behave correctly only in such cases. The demonstration vote-stealing program developed by Feldman et al. [35] behaves exactly this way, suppressing malicious behavior when data on the machine indicates that testing is underway. A software bug may of course have similar behavior, since faulty DRES may behave arbitrarily. For example, a bug that only manifests when many votes have been cast or several hours have elapsed is unlikely to be identified using L&A test procedures. We must also take care that a malicious poll worker cannot signal the testing condition to the voting machine using a covert channel such as a “secret knock” of user interface choices.

Given this capacity to “lay low” under test, the problem of fooling a voting machine into believing it is operating in a live vote-capture environment is paramount [60]. Because L&A testing commonly makes explicit use of a special code path, parallel testing is the most promising scenario. It presents its own unique hazard: if the test successfully simulates an election-day environment, any votes captured under test will be indistinguishable from legitimate ballots cast by real voters, so special care must be taken to keep these votes from being included in the final election tally.

2.8 Cryptography and e-voting

Many current commercial DRES attempt to use encryption to protect the secrecy and integrity of critical election data; the many studies and audits described in Section 2.1 have demonstrated that these attempts are universally unsuccessful.

A first step is to encrypt votes to protect them from prying eyes after they are cast and before they are counted; this is a straightforward analogy to the physical security of the voting booth and the ballot box in conventional paper voting. Unfortunately, this is insufficient for voter verifiability: how does she know that her ballot was encrypted correctly (or, indeed, was correct at the time it was encrypted)? How can ballots be safely decrypted during the tally process, and how can the tally be confirmed to be accurate?

Researchers have recently developed a number of sophisticated cryptographic techniques to address these problems in the voting domain. Some assist with aspects of verifiability, others integrity, and still others privacy. The major techniques in the field are summarized below.

Mixnets. One line of research has focused on encrypting whole ballots and then running them through a series of “mix nets” (adapting work by Chaum originally for anonymous email [20]) that will re-encrypt and randomize ballots before they are eventually decrypted (see, e.g., [83, 74]). If at least one of the mixes is performed correctly, then the anonymity of votes is preserved. This approach has the benefit of tolerating ballots of arbitrary content, allowing its use with unconventional voting methods (e.g., preferential or Condorcet voting). However, it requires a complex mixing procedure; each stage of the mix must be performed by a different party (without mutual shared interest) for the scheme to be effective.

Homomorphic tallying. The encryption system allows encrypted votes to be added together by a third party without decrypting them; the individual vote plaintexts are therefore concealed during tabulation. Many ciphers, including El Gamal public key encryption, can be designed to have this property. See Chapter 4 for the application of this technique in VOTEBOX.

Zero-knowledge proofs. In any voting system, we must ensure that votes are well formed. For example, we may want to ensure that a voter has made only one selection in a race, or that the

voter has not voted multiple times for the same candidate. With a plain-text ballot containing single-bit counters (i.e., 0 or 1 for each choice) this is trivial to confirm, but homomorphic counters obscure the actual counter's value with encryption. By employing zero-knowledge proofs (particularly non-interactive zero knowledge proofs, or NIZKS [14]), a machine can include with its encrypted votes a proof that each vote is well-formed with respect to the ballot design (e.g., at most one candidate in each race received one vote, while all other candidates received zero votes). Similarly, NIZKS can be used to prove (without disclosing the decryption key) that a homomorphically summed vote total is decrypted correctly. The attached proof is *zero-knowledge* in the sense that the proof reveals no information that might help decrypt the encrypted vote. Note that although NIZKS like this can prevent a voting machine from creating invalid ballots or stuffing the ballot box, they cannot prevent a voting machine from flipping votes from one candidate to another (cast as intended).

Bulletin boards. A common feature of many cryptographic voting systems is that all votes are posted for all the world to see. Individual voters can then verify that their votes appear on the board (e.g., locating a hash value or serial number “receipt” from their voting session within a posted list of every encrypted vote). Any individual can then recompute the homomorphic tally and verify its decryption by the election authority.

Blind signatures. Originally developed by Fujioka, Okamoto, and Ohta (FOO) [39] and implemented in the Sensus [29] and EVOX [47] voting systems, blind signatures allow election administrators to issue cryptographically official but “blank” ballots to voters. The election administrator has not seen the voter's choices at the time of the signature, hence they are “blind.” Only registered voters can obtain blind signatures, so only they can vote; these signatures may only be used once, preventing duplication of the credential. In the FOO protocol, once a voter has completed the ballot, it is then sent (unblinded) over an anonymous channel to the tabulator.

2.9 “Paper cryptography”

In response to the difficulty in explaining cryptography to non-experts (and, in some cases, as an intellectual exercise), scholars have designed a number of cryptographic paper-based voting systems that have end-to-end security properties. They do not in general apply “traditional” numerical cryptography (RSA, El Gamal, and the like) but they are information-theoretic techniques nonetheless and will be referred to here as “paper cryptography.” By transporting cryptographic approaches out of the realm of abstract algebra and into the physical domain, it is hoped that the result will be less frightening to users but no less powerful.⁶

The earliest such approach was developed by Chaum [21]; it uses visual cryptography (invented by Naor and Shamir [73]) to create a printed ballot comprising two partially-transparent parts that, when overlaid, reveal the voter’s choices, but reveal nothing at all in isolation. One half of this ballot is retained by the voter as a receipt; an imprinted identifier allows her to consult a bulletin board to ensure that her choices were tallied (via a re-encryption mixnet) correctly.

Prêt à Voter [25, 82] also uses a separable ballot design to allow the voter to leave the polling place with a receipt that proves the existence of her ballot in the tally (but not the exact vote cast); it achieves this property by randomizing the order of candidates. In order to cast a ballot, the voter’s receipt is scanned at the polling place and stored for later tabulation. The ballot, which preserves the voter’s privacy should she be forced to reveal it, is simply a list of checked boxes with the (randomized) candidate names removed. The receipt also contains an encrypted value; only election trustees are able to use this value to reconstruct the correct order of the candidates and thereby count the ballot.

Punchscan [38], a subsequent design by Chaum et al., also uses a physical ballot design that behaves as a two-share cipher thanks to randomized candidate ordering. A Punchscan ballot comprises two overlaid (opaque) sheets. The top layer is imprinted with candidate names, each associated with a letter code; there is also a separate set of holes, one per candidate. The bottom layer contains the same letter codes, positioned to be visible through the holes when the top layer is overlaid. The voter casts her ballot by using a bingo dauber.⁷ When applied to a hole in the two-layer ballot, the dauber

⁶Contrast this with the idiomatic Chinese 纸老虎 (“paper tiger”), a fearsome-seeming but ultimately toothless beast.

⁷A bingo dauber is an ink marker with a very large tip; it is used for convenient marking of cards in the lottery game Bingo.

creates a sufficiently large mark that some ink is left on each layer. The code assignment on the top layer, and the order of codes on the bottom layer, match one another but are chosen randomly for each ballot; therefore, once separated, neither layer contains enough information on its own to reveal the voter's choices. Each half retains a serial number, however, and with this election officials are able to reconstruct the voter's choices using just one half of the ballot (using a database mapping serial numbers to candidate code assignments). A voter therefore scans one half of her completed ballot at the polling place and takes it home as a receipt; the other half is destroyed. While she cannot use this receipt to prove to anyone how she voted, she can ensure that her ballot was received (cast-as-intended) by confirming that her receipt is present on an official bulletin board where all scans are posted. She can also verify the correctness of the tally (counted-as-cast) by auditing a portion of the tabulation process.

This line of research has culminated in Scantegrity [23], which uses a single ballot sheet with tear-off strips that separate the encrypted shares; and Scantegrity II [22], which associates with each candidate a confirmation code that is revealed by invisible ink during the ballot marking process. The latter is designed to resemble as closely as possible the current optical-scan voting process. It has the distinction of being the first paper-cryptography voting system to be selected for use in a civic election: the city of Takoma Park, Maryland plans to use Scantegrity II in its municipal elections in November 2009 (a mock election to test the system is imminent as of this writing).

ThreeBallot [78, 79] requires a voter to create three different ballots, two of which correspond to her true choice, and one of which is the *opposite* of her choice. One of these ballots is duplicated as a receipt, which the voter can then check against a public bulletin board of votes cast to confirm (with probability $\frac{1}{3}$) that her vote is intact.

The first difficulty besetting these paper-crypto approaches is one of usability. There are accessibility implications of schemes that necessarily revolve around paper (Punchscan, Scantegrity, Prêt à Voter, ThreeBallot): voters with disabilities that prevent them from marking paper will be unable to use the paper-based systems without assistance.

In general, despite being nominally paper-based, they are substantially more complex (and likely confusing) than conventional paper balloting systems. Early indications are that such additional

measures will increase unacceptably the burden on the voter to understand the system in order to correctly and confidently cast a ballot. Unusual ballot designs must be explained to voters (and, equally importantly, poll workers) to ensure that each voter is able to vote correctly. In ThreeBallot, voters who do not correctly mark their ballots (including the counterintuitive “opposite vote” described above) can cause overall vote totals to be unrecoverably flawed.

More seriously, however, these approaches may not be legal in many jurisdictions. For example, they commonly rely upon candidates appearing on the ballot in randomized order; this would be forbidden in Texas, where state statute requires candidates to appear in a specific order.⁸

2.10 Paper plus computers: Optical scan and vVPAT

Optical-scan voting systems, in which the voter marks a piece of paper that is both read immediately by an electronic reader/tabulator and reserved in case of a manual audit, achieve the *cast as intended* property at the cost of sacrificing some of the usability and logistical benefits afforded by DRES.

The voter-verifiable paper audit trail (vVPAT) allows a DRE to *create* a paper record for the voter’s inspection and for use in an audit, but it has its own problems. Adding printers to every voting station dramatically increases the mechanical complexity, maintenance burden, and failure rate of those machines. A report on election problems in the 2006 primary in Cuyahoga County, Ohio found that 9.6% of vVPAT records were destroyed, blank, or “compromised in some way” [48, p. 93].

Even if the voter’s intent survives the printing process, the rolls of thermal paper used by many current vVPAT printers are difficult to audit by hand quickly and accurately [41]. It is also unclear whether voters, having already interacted with the DRE and confirmed their choices there, will diligently validate an additional paper record. (In the same Cuyahoga primary election, a different report found that voters in fact did not know they were supposed to open a panel and examine the printed tape underneath [4, p. 50].)

⁸The particular order depends on the candidate’s party affiliation. Candidates affiliated with political parties are listed in order of the votes received by each party in the prior gubernatorial election [1]; unaffiliated candidates appear in an order chosen by random lottery [2]. In each case, the order is fixed for all ballots cast in the state.

2.11 Internet voting

The question of voting over long distances via the Internet is recurrent. Fundamental problems of voter privacy and equipment trustworthiness (described in more detail in Chapter 8) plague efforts to provide Internet voting for any serious election.

Nonetheless, attempts have been made. A recent example: The U.S. military planned to deploy in 2004 an Internet-based electronic voting system called the Secure Electronic Registration and Voting Experiment (SERVE). They convened a panel of experts to evaluate the proposed system; a subset of those experts wrote a report describing all of the problems with voting over the Internet, such as easily compromised client platforms [55]. The military canceled the program, replacing it with a fairly simple fax-based scheme that is arguably even less secure than SERVE [54].

In the U.S., several “primary” elections have been conducted over the Internet, including the recent “Democrats Abroad” primary election. Standard web browsers on standard client computers were used, and no particular measures were taken (or really could have been taken) to prevent voter bribery and coercion, much less deal with viruses or worms that might try to compromise the browser’s behavior. In fact, the Democrats Abroad’s primary did not have a secret ballot. In a radio interview⁹, the administrator of the election said that the votes were actually public. The official disseminated results¹⁰ only present country-by-country subtotals, so it’s unclear exactly how much privacy is granted to Democrats Abroad’s voters.

Internet voting has been used, perhaps more successfully, in national elections in Estonia [98]. The user authentication builds on a national ID card which contains a smart-card chip. Prospective voters insert the card into their computer, with a suitable adapter, and it allows them to authenticate to a government web site over an SSL-encrypted channel where they may cast a vote. Voters may vote as many times as they like, with the final one actually being tallied. The ability to cast multiple votes provides some limited resistance against bribery and coercion attacks. The use of SSL provides resistance against network man-in-the-middle attacks. Nothing in the Estonian voting architecture provides any protection against compromised client platforms.

⁹<http://weekendamerica.publicradio.org/display/web/2008/01/25/demsabroad/>

¹⁰<http://www.democratsabroad.org/sites/default/files/DA%20Global%20Primary%20Results%20FINAL%20REVISED.pdf>

Among commercial DRE voting systems, several vendors allow the use of modems to transmit election results (insecurely). While some states ban the use of these modems, others allow them under the guise of “unofficial” early election results. While this ignores the risk that an attacker may be able to compromise the tabulation system by calling it up on the telephone, these states are assuming that the records stored in the DRE systems themselves will survive the interval between the end of the election and their return to the voting warehouse, after which electronic results can be extracted directly from the voting machines. A similar property holds for bulletin boards, which can be disseminated in any way that data can be transmitted.

Helios [5] is a Web-based system that sacrifices coercion resistance for a verifiable and minimally complex crypto-voting system that can be used from a voter’s home computer. It employs homomorphic encryption, allowing a simplified variant of Benaloh’s ballot challenge [10] with a single trusted server maintaining a bulletin board for cast ballots. A mixnet is used for privacy-preserving decryption. Helios is intended for “low-coercion elections,” but if used exclusively in supervised remote polling places it could be suitable for high-stakes national contests as well.

Civitas [27] is an ambitious cryptographic voting system designed to allow Internet-based voting on a large scale. It too employs homomorphic encryption, mixnets, and a bulletin board. It suffers some limitations that preclude its straightforward deployment in nationwide elections, notably the requirement that each voter be issued a long-term cryptographic key pair for the purpose of acquiring per-election voter credentials. Moreover, an explicit design goal of the Civitas work is allowing *unsupervised* Internet voting. The authors admit that this requires trust in the end user’s computer, and they respond to this by suggesting that voters seek out a voting terminal that they trust (e.g., one maintained by a political party or social organization). This proviso causes a *practical* Civitas deployment to look quite a bit like the remote polling places described in Chapter 8.

Adder [64] is another Internet-scale cryptographic voting system, incorporating the (now-familiar) techniques of homomorphic tallying and bulletin boards. Like Civitas, Adder requires trusted hardware; the authors of Adder also admit that the system does not support methods for the voter to verify the correct operation of the equipment and that the database required to maintain the system is vulnerable to tampering or loss. The techniques described in the following chapters address these

problems (among others) directly.

2.12 Byzantine faults in distributed systems

As Chapter 3 will show, a VOTEBOX polling place is a distributed system. In the distributed systems community, a distinction is drawn between simple, detectable, obvious failures (the *fail stop* model) and failures that cause participating entities to behave arbitrarily. This latter case, termed the *Byzantine failure model*, allows for faulty nodes to continue operation but to do so incorrectly, and even to present inconsistent data to other nodes in the system. Such a model is used by security researchers to encompass malicious behavior, and in the case of electronic voting we may consider either a malicious or simply buggy voting system to be a Byzantine failure.

In Lamport's original definition of this failure model [68], he shows that a distributed system whose goal is state replication across nodes can tolerate up to one-third of those nodes in a Byzantine failure state. More formally, to tolerate f faulty nodes, a system must be comprised of $n \geq 3f + 1$ nodes. Byzantine fault tolerance (BFT) in this case is defined as overall correct operation of the system (that is, the system reaches the correct state, or equivalently provides the correct answer to some client) in the face of f nodes disrupting the system's operation arbitrarily.

The state-machine replication perspective is a powerful one, encompassing many deterministic networked services, and Practical Byzantine Fault Tolerance [19] shows how such failures can be recovered efficiently in an asynchronous system. More recently, the PeerReview [43] system leverages the determinism of a state machine replication system to identify faulty nodes by probabilistic audits; auditors, provided with the node's inputs, run a reference implementation to confirm that the node's outputs are correct. Regrettably, BFT techniques do not apply in the voting realm for exactly this reason; the inputs to the system are *not deterministic* (incorporating cryptographic randomness as shown in Chapter 4) and the inputs are *not available* for replay (because the inputs, in this case, are the voter's actions in the booth, which must remain secret).

CHAPTER 3

AUDITORIUM

3.1 Introduction

While Chapter 4 will address the issue of *correctness* of voting machines, this chapter presently considers a different facet of the election problem, inspired by a real-world experience with a contested election involving DRE systems.

When the results of an election are in doubt, the usual course of action is to perform a recount. When DRES are used by the electorate, the data surveyed during such a recount is purely electronic, and hence, fundamentally mutable. Any party in possession of the machine or its flash memory cards or the tabulation system might be able to alter or destroy votes. What does it mean to recount votes whose provenance cannot be proven?

The problem extends beyond ballots. It is common for DRE machines to keep an *event log* to support *post facto* analysis when the correct operation of the machines comes into question. These logs, which record and timestamp interesting events such as “election started” or “ballot cast,” can provide critical clues to the events of election day, especially when the vote tally is unusual or inconclusive. That is, of course, unless they’ve been tampered with, in which case they prove nothing at all.

What would be required for electronic voting machine event logs to serve as robust forensic documents? They must stand up to scrutiny during an audit, even under the assumption that the voting machines may have been tampered with, damaged, or lost at any point from their manufacture until the last recount. Specifically, they should describe a provable timeline of valid events, including

administrative steps and votes cast, transpiring on election day.

In this chapter I detail the design of Auditorium, a secure logging and networking facility that I developed to address the problems of survivability and integrity in voting system audit logs. This work is inspired by a unique opportunity that presented itself in 2006: an invitation to investigate a contested election.

3.2 March, 2006: Laredo, Webb County, Texas

The citizens of Webb County, voting in the 2006 primary,¹ were given the option to vote either on paper (optical-scan) ballots or using the county's new iVotronic touch-screen DRE systems, manufactured by Election Systems & Software (ES&S). In this particular election, the second-place finisher in a local judicial race found that he received a smaller share of the DRE vote than the paper vote, and so contested the electronic election results. As a result, the machines used in the election were taken into custody on orders of the judge in the case, who also permitted the challenger to find independent voting security experts to help with the investigation.

Dan Wallach and I were contacted and made two trips to Laredo, the seat of Webb County, to examine the impounded and impugned machines (which can be seen in Figure 3.1). Although we did not have access to source code (nor any form of assistance from ES&S) we were able to examine the machines themselves, the memory cards used to copy results from the machines to the tabulator (a general-purpose computer running special software), and the files output by that tabulator.

Such access, while not without precedent, was at the time² still a rare opportunity for e-voting researchers, who had theretofore had a rather hostile relationship with commercial voting system vendors; in short, it was difficult to get access to voting systems to perform any kind of analysis. More rare still was a situation in which analysis could be performed on voting systems that had recently been used in an election under dispute; we were fortunate to be in a position to participate in this “live fire” scenario. We submitted our final report [101] to the court in May of that year.

¹A primary election is a pre-election nominating contest peculiar to the United States; voters of a particular political party are given the opportunity to choose their candidate from a number of contenders of the same party.

²The Ohio and California voting system reviews of 2007 (see Chapter 2) provided researchers with source code, technical documentation, and hardware access.

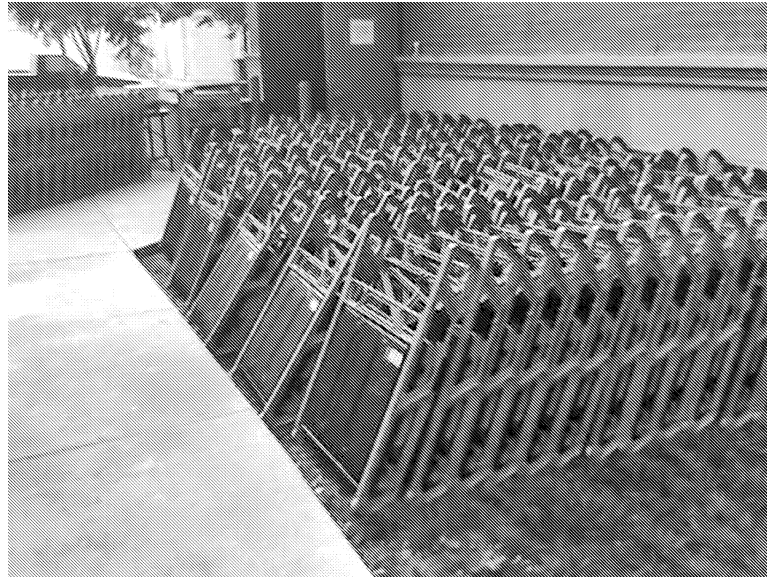


Figure 3.1: Impounded voting machines in Laredo, TX. Approximately half of the ES&S iVotronic machines used in the Webb County primary election can be seen stacked outside the door to the storage facility beneath the county courthouse; the rest of the machines were still inside, waiting to be extricated for inspection. (Photo taken April 25, 2006, 10:28 AM.)

3.2.1 Findings

We found no direct evidence of tampering in the ES&S machines used in Webb County, nor did we have the ability to examine their source code for faults. We did, however, discover anomalous and incomplete information in the *event logs* kept by the iVotronic machines. These logs, stored in a proprietary format on the flash memory inside the voting machines, exist to provide some degree of auditability after the election is over. When the polls close, poll workers copy the contents of the voting machines onto CompactFlash memory cards, which are then transferred to a general-purpose PC running the ES&S tabulation software. An example of the tabulator's output, when given as input one machine's binary event log, is shown in Figure 3.2. We examined both the text logs emitted by the tabulation software and the raw binary logs stored on the machines themselves.

Votronic	PEB#	Type	Date	Time	Event
5140052	161061	SUP	03/07/2006	15:29:03	01 Terminal clear and test
	160980	SUP	03/07/2006	15:31:15	09 Terminal open
			03/07/2006	15:34:47	13 Print zero tape
			03/07/2006	15:36:36	13 Print zero tape
	160999	SUP	03/07/2006	15:56:50	20 Normal ballot cast
			03/07/2006	16:47:12	20 Normal ballot cast
			03/07/2006	18:07:29	20 Normal ballot cast
			03/07/2006	18:17:03	20 Normal ballot cast
			03/07/2006	18:37:24	22 Super ballot cancel
			03/07/2006	18:41:18	20 Normal ballot cast
			03/07/2006	18:46:23	20 Normal ballot cast
	160980	SUP	03/07/2006	19:07:14	10 Terminal close

Figure 3.2: An iVotronic event log. The machine in question has the serial number 5140052; several different PEBs (special administrative access tokens) were used over the course of the day. Noteworthy: the machine was cleared and entered into service at about 3:30 PM on election day.

Lost votes

Figure 3.2 shows something unexpected. While polls opened for the primary election around 7 AM on March 7, 2006, this particular machine was cleared and entered into service at about 3:30 PM that same day. This could be entirely innocuous: perhaps this machine was simply unneeded until the early afternoon, at which point poll workers activated it.

However, the machine might also have been accepting votes since 7 AM like the other machines in that precinct, but was wiped clean in the afternoon. Because the machine is trusted to keep its own audit and vote data—both of which can be erased or otherwise undetectably altered—we cannot be sure that votes were not lost.

There exists a procedure to mitigate against this sort of ambiguous vote record, albeit a fragile one. Official election procedures direct poll workers to print a “zero tape” on each machine before it is entered into service on election day, and a “results tape” once the polls are closed. Each tape reveals (in addition to the election-specific, per-race tallies) the contents of the machine’s *protected count*, a monotonic counter inside the machine that is incremented any time a ballot is cast and, according to the manufacturer, cannot be decremented or reset even if the machine is cleared. This is a feature first found on mechanical (lever-based) voting machines.

It should therefore be possible (assuming the counter resists tampering) to compare the difference in the count on the zero and result tapes and the number of ballots recorded on that machine in between. If the numbers are not equal, votes cast on election day were lost, or votes cast on other days are being treated as legitimate, or both.

Unfortunately, the system does not require that these tapes be printed, nor that they be properly stored. In the case of machine 5140052, we were unable to locate a zero tape; the results tape showed a protected count of 12, and we observed 6 votes in the final tally from that machine, so a maximum of 6 votes were lost. It is quite possible that *no* votes were lost, and that the other 6 votes were votes cast at other times for other purposes (e.g., other elections or tests). We cannot be sure, and had the machine been in service for many years, its protected count would be much higher, correspondingly inflating our best upper bound on the number of lost votes.

Other anomalies and ambiguities

We encountered several machines whose logs attest that votes were cast on those machines on or before March 6, the day *before* the primary election. Some of these machines showed what appeared to be a normal voting pattern, with the exception that every vote was cast on the 6th. Inspection of those machines (an example is shown in Figure 3.3) revealed that their hardware clocks were off by one day, implying that the votes in question were in fact cast during the election on the 7th. We do not know for sure; anyone with access to the machines prior to the election could have cast these ballots illegitimately.

Other machines (e.g. Figure 3.4) with votes cast on the wrong day fit a different pattern. In each case, two votes were recorded: one ballot in the Republican primary election and one in the Democrat primary. For each ballot, the particular candidates chosen were the same each time. We learned that this is the profile of a machine under “logic and accuracy” test; election officials would cast a couple of ballots and satisfy themselves that the machines were working. Somehow these test votes were being counted in the official election tally.

Votronic	PEB#	Type	Date	Time	Event
5142523	161061	SUP	02/26/2006	19:07:05	01 Terminal clear and test
	161115	SUP	03/06/2006	06:57:23	09 Terminal open
			03/06/2006	07:01:47	13 Print zero tape
			03/06/2006	07:03:41	13 Print zero tape
	161109	SUP	03/06/2006	10:08:26	20 Normal ballot cast
			03/06/2006	12:39:05	20 Normal ballot cast
			03/06/2006	14:49:33	20 Normal ballot cast
			03/06/2006	15:59:22	20 Normal ballot cast
			03/06/2006	18:01:45	20 Normal ballot cast
			03/06/2006	18:10:24	20 Normal ballot cast
			03/06/2006	18:26:52	20 Normal ballot cast
			03/06/2006	18:29:18	20 Normal ballot cast
			03/06/2006	18:39:41	20 Normal ballot cast
			03/06/2006	18:44:24	20 Normal ballot cast
	161115	SUP	03/06/2006	19:29:00	27 Override
			03/06/2006	19:29:00	10 Terminal close

Figure 3.3: Machine showing the wrong date. These votes appear to be cast on the day *before* the election; when we inspected the machine we found that its hardware clock was off by one day, implying that these are likely to be valid election-day votes.

Votronic	PEB#	Type	Date	Time	Event
5145172	161061	SUP	03/06/2006	15:04:09	01 Terminal clear and test
	161126	SUP	03/06/2006	15:19:34	09 Terminal open
	160973	SUP	03/06/2006	15:26:59	20 Normal ballot cast
			03/06/2006	15:30:39	20 Normal ballot cast
	161126	SUP	03/06/2006	15:38:37	27 Override
			03/06/2006	15:38:37	10 Terminal close

Figure 3.4: Likely test votes. This machine also shows votes cast on March 6, the day before the election. When we inspected this machine, however, its hardware clock was set to the correct date.

Administrative and procedural mistakes

We also saw evidence of procedural failures which call into question the accuracy of the vote tally. In Figure 3.2, the event described as *Super ballot cancel* represents a situation where a “supervisor” (poll worker) had to abort an in-progress voting session. This typically happens when voters “flee,” that is, they leave the polling place without completing a ballot. In this event, poll workers are under instruction to *cancel* the incomplete ballot and to *record* on a paper log the reasons for having done so.

These paper logs were rarely kept in this particular primary election, so we have no way to confirm the legitimacy of the cancellation. (It is not possible to cancel a vote after it has been successfully cast.)

We conclude from these experiences that electronic voting systems generate a great deal of auditing data that can shed light on irregular results. Because the data for the entire county was available in digital form, we were able to analyze a large amount of election auditing data at great speed, a feat that would have been far more difficult if we relied on paper records kept in each precinct by poll workers. Despite their usefulness, however, we are still able to prove neither the accuracy nor completeness of these event logs.

3.3 The design of Auditorium

3.3.1 Requirements for auditable voting systems

The first step in designing a voting system which survives mistakes and failures to provide an unambiguous result is to formalize the required properties of such a system. In an *auditable voting system*:

R1 Each machine must be able to account for every vote. Any ballot to be included in the final tally must be legitimate; that is, it must provably have been cast while the polls were open. It must also be possible to prove, by examining the auditing records, that no legitimate votes have been omitted from the tally. This property should extend beyond votes to other important events, such as ballot cancellation.

R2 A machine's audit data and cast ballots must survive that machine's failure. The overall system must defend against the loss of critical election data due to malfunction, loss, destruction, or tampering with individual machines.

R1 and R2 are sufficient to detect and recover from the procedural errors that were observed in Laredo and that can cast doubt upon even legitimate election results. The next section describes the way Auditorium meets these challenges.

3.3.2 How I learned to stop worrying and love the network

The solution presented here to the problems of resilient, believable audit records revolves around the idea that **auditability can be enhanced by connecting voting machines to one another**. The general idea of networking the polling place is not original; some electronic voting systems (the Hart InterCivic eSlate, for one) already support the use of a network. However, the elections community has historically been very suspicious of networks, and with good reason: any unjustified increase in the potential attack surface of a voting machine is inexcusable.

A network could make possible two kinds of previously-infeasible attacks: voting machines could be attacked from outside the polling place, and a single compromised voting machine can now attack others from the inside. If a polling place is networked, it must *not* be reachable from the Internet, obviating an outside attack. Such an “air gap” is already an important part of military computer security practices and is sensible for electronic voting.

The “inside attack” is an interesting case. An attacker needs physical access to only one machine (perhaps the one on which the attacker is voting) in order to install malicious code, which can then spread via the network. Note that the network is not necessary for this kind of attack. One of the chief features of DRE voting machines is the speed with which they may be tallied; this speed comes from some sort of communication between machines, whether in the form of a network or simply exchanged memory cards. The Diebold AccuVote-TS system uses flash memory cards for this purpose and Feldman et al. [35] found that this card-swapping was an effective way to spread a “voting machine virus.” Yasinsac et al. [103] found a similar vulnerability with the ES&S system. In the end, the lack of a network does not guarantee isolation of any faulty or malicious voting machine.

It is true that a networked voting system is strictly *more* vulnerable than one that does not have a network, by virtue of the additional complexity and potential ingress points it creates. The following sections detail the security properties that justify this additional risk.

3.3.3 Secure logging

I now build up a design for an auditable voting system from essential building blocks, of which the first is *secure logging*. The requirement R1 would be satisfied by a voting machine able to produce a

tamper-evident record of ballots cast and other pertinent events. A first step is a *secure log*, such as those described by Bellare and Yee [9] and Schneier and Kelsey [91]. Each event in a secure log is encrypted with a key that is thrown away so that, if attackers gain control of a machine, they should be unable to read log messages written in the past (that is, before the attack). Encryption keys are generated deterministically from one another, starting with an initial key that is retained by a trusted party. To read the logs, the sequence of keys can be re-generated, and log entries decrypted, given the initial key.

The inability of untrusted parties to read previous log entries, termed *perfect forward secrecy*, is not necessary for electronic voting, where the pertinent data is a public record. Instead, the record needs *forward integrity* [9], the property that an attacker may not undetectably remove, add to, or alter auditing records committed before the attack. This can be achieved with *hash chaining*. Each event E_i includes $\text{hash}(E_{i-1})$, the result of a collision-resistant cryptographic hash function applied to the contents of the previous event.

If the contents of E_{i-1} are hard to predict (for example, it includes a *nonce*: a random ephemeral value), the time at which E_i was committed to the log is now *backward-constrained*: it must succeed the time of E_{i-1} . When event E_{i+1} in turn incorporates the hash of event E_i , E_i is now *forward-constrained* as well. Thus, each event E_i , containing log data d_i , has the form $[d_i, \text{nonce}, \text{hash}(E_{i-1})]$. (Naturally, there must exist a special event E_0 from which the first real event E_1 derives; it can be defined as a well-known arbitrary value, such as a string of zeros of appropriate length.)

3.3.4 Entangled timelines

Moving beyond the realm of a node's own timeline, let us now consider ways to reason about multiple timelines in a distributed system (such as a polling place). The concept of fixing events from foreign, untrusted timelines in the reference frame of local events originates in the logical clocks of Lamport [67]. Maniatis and Baker make this scheme tamper-proof by fusing it with hash-chaining to form what they call "timeline entanglement" [69]. An *entangled timeline* is a secure log which includes, among the links in its hash chain, events from the secure logs of other (possibly untrusted) parties. Alice might, for example, send an event to Bob, who can now mix that information into *his*

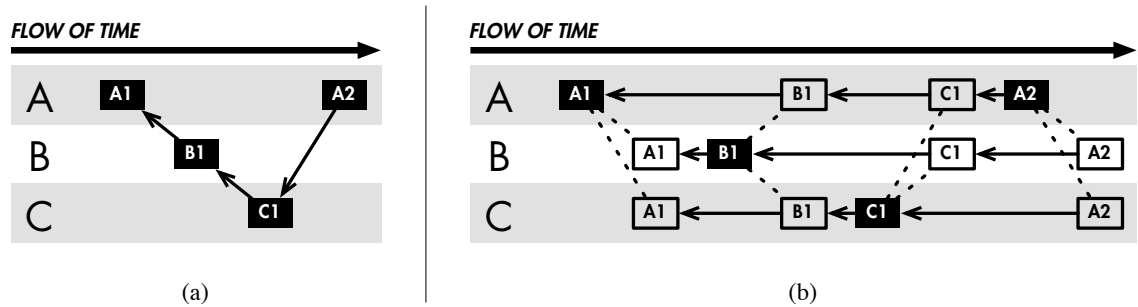


Figure 3.5: Flow of time in the Auditorium. Participants *A*, *B*, and *C* experience a shared flow of time (a). Solid arrows denote direct temporal precedence; for example, $\mathbf{x} \leftarrow \mathbf{y}$ illustrates an event *Y* that incorporates $\text{hash}(X)$, proving that event *X* must precede event *Y* in time. The abstraction of a shared timeline is achieved by replicating local events to remote nodes (b); dotted lines represent Auditorium broadcasts.

next event. Now Bob can prove to Alice that his event succeeds hers in time.

Much like including a copy of today’s newspaper in a photograph to fix it in time for any skeptical auditor, entangled timelines allow parties to fix the events of others in their own timelines. By following links of hash chains to and from a foreign event in question, a node should be able to eventually reach events in its own timeline that provably precede and succeed it.

3.3.5 Auditorium: n-way entanglement and replication

With entangled timelines, the system is able to satisfy requirement R1 without the need of either a trusted auditor holding a base key or a trusted timestamping service. The burden of R2, however, remains unmet; the system can detect erasure, but not recover the records.

To this brew of concepts I therefore add insight from peer-to-peer research: in a world where disk and network are cheap and abundant, one has the luxury of widespread replication. While debate continues as to whether these criteria hold in the wider Internet, I argue that it is certainly true for electronic voting: even the low-end computers used in DRES are overprovisioned for the task of voting.

I introduce a simplification of the entangled timeline scheme that is practical for small networks such as a group of voting machines in a polling place. Rather than periodically exchanging a fraction of their events along with hash-chain precedence proofs (as in Maniatis and Baker), nodes in this

system will *broadcast* every state change that one will ever want to reason about or recover. This composition of techniques forms the kernel of the system, called Auditorium: Every event is heard and recorded by every participant, and each new event is entangled with events from the past. The resulting abstraction—a single shared and tamper-evident record of time—is illustrated in Figure 3.5.

Every event is cryptographically signed by its originator to prevent forgery. The use of signatures in conjunction with entanglement prevents later repudiation of events, or even repudiation of the time at which those events occurred. A log entry E_i from node A therefore has the form $A.\text{sign}([d_i, \text{nonce}, \text{hash}(E_{i-1})])$, where $A.\text{sign}(X)$ denotes X signed with A 's public key.

As a part of the simplification, Auditorium foregoes the multiple-phase protocol used by Maniatis and Baker to exchange precedence proofs; the result is that the shared timeline may naturally diverge due to asynchrony. For example, Bob and Charlie may both publish events including a hash of Alice's last message. These two events are *contemporaneous*; that is, they cannot be distinguished in time because they directly succeed the same event. If Alice wants to broadcast another event, she has a choice: should her event include the hash of Bob's event or Charlie's? She must choose *both* in order to forward-constrain both events in time. Therefore, her message should *merge the timelines* by including the hash of any prior event not already constrained. An event E_i may therefore include the hashes of any number of prior unconstrained events:

$$E_i = A.\text{sign}([d_i, \text{nonce}, [\text{hash}(E_j), \text{hash}(E_k), \dots]])$$

3.3.6 The Auditorium broadcast network

To create the abstraction of a broadcast channel, the implementation of Auditorium used in the VOTEBOX prototype voting system uses a fully-connected network of point-to-point TCP sockets. Every participating node in a network of size n is connected to every other, resulting in a complete graph (K_n) of connections. Any new message generated by a participant should be sent on every open connection. To illustrate: if Alice, Bob and Charlie comprise an Auditorium instance and Alice wishes to announce an event to the Auditorium, she sends her message to Bob and Charlie.

Furthermore, nodes should exchange messages according to an epidemic or gossip protocol; any

node receiving a “new” event (one whose hash it has never before seen) should forward it to every other participant. In the above example, Bob and Charlie would each forward the new event to the other; upon receiving what is now an “old” event, no more forwarding would take place.

Such a network, quadratic in the number of open connections, is hardly the only way Auditorium could have provided the broadcast abstraction, and is certainly not the most scalable. For the problem of voting, however, the number of nodes involved is quite small (see Section 7.4). This mechanism is extremely robust and simple to implement; no complex tree-construction algorithms or maintenance operations are required; every node is simply connected to every other, and shares new messages with them all.

Because new messages are flooded on every link, nodes hear about the same message from every other node in the system. This flurry of seemingly-redundant traffic has the extremely valuable benefit of providing robustness to Byzantine faults in timeline entanglement [69]; a node might attempt to reveal divergent timelines to different neighbors, but in the Auditorium this duplicity will be quickly exposed as conflicting messages are exchanged among their recipients.

Perhaps most importantly, the extreme redundancy inherent in this design provides a believable account of the entire election, including sufficient information to authenticate and tally each ballot, even if $n - 1$ nodes fail after election day. This is a distinct possibility given that n may be as small as 2. Many jurisdictions that use paper ballots, for example, will require precincts to maintain one DRE for accessibility reasons. If this one DRE were a VOTEBOX, when combined with its supervisor console, it forms an Auditorium network of 2. In a larger precinct, tolerating $n - 1$ failures is still important, given that an insider may attempt to modify or destroy one or more machines after the election. With Auditorium as described, just one honest VOTEBOX is all that is needed to act as a “whistleblower” and recover all the true votes from election day. (To know which machine to trust, an auditor must rely on the true polls-closed hash from the end of the election; see Section 3.4.2.)

To see how effective this approach is, consider the broader design space in which each VOTEBOX trades off some amount of robustness to reduce the bandwidth demands of the system. Figure 3.6 shows simulation runs using a synthetic election-day workload (described more fully in Section 5.4.3). The number of replicas r (redundant machines to which any given message is sent) varies from 1 (no

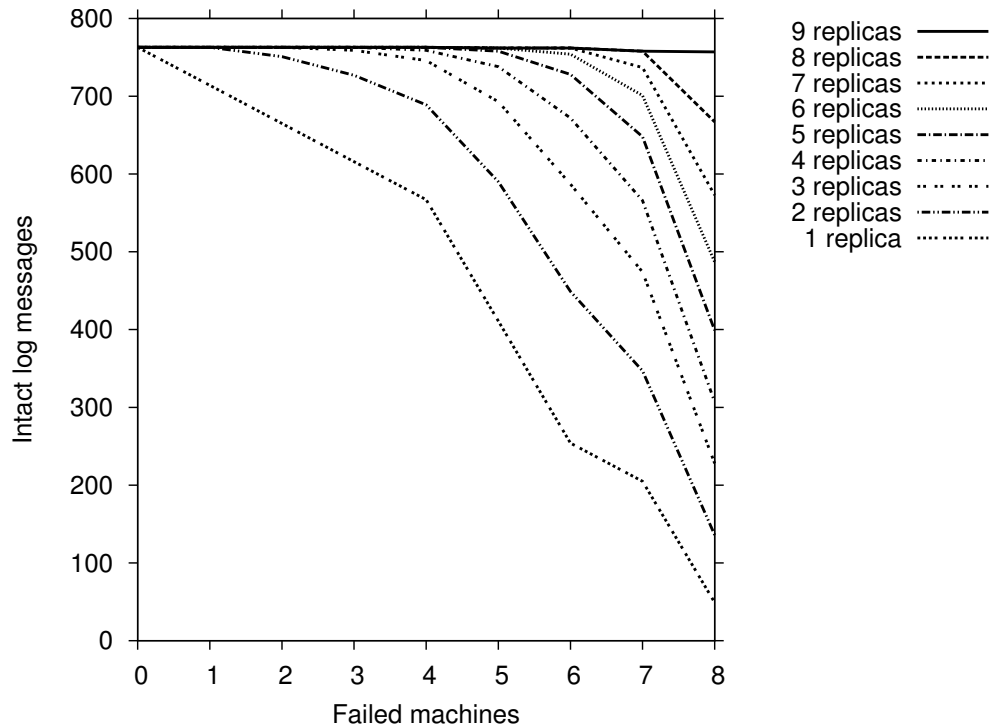


Figure 3.6: Data survival under various replication factors. As the number of “failed” machines (either totally destroyed or compromised in some way as to cast doubt on the integrity of its data) increases, the number of Auditorium messages that can be recovered from the remaining machines decreases. Data loss is mitigated by adding replicas; with 9 replicas, even a loss of 8 nodes can be completely tolerated. (The 9-replica line does indeed nudge downward slightly in the 8-failures case; this is due to initial bootstrapping messages that were never heard by the remaining replica.)

replication: each machine keeps its own logs) to n (the Auditorium full replication strategy). Points are the average of three runs with different random choices of which r machines to replicate with for each message, and which f machines are chosen to fail. As expected, only with full replication ($r = n$) can the system tolerate many failures without loss.

3.3.7 Voting in the Auditorium

As I have described it thus far, Auditorium is a general-purpose auditable group communication system; a specific protocol must now be specified for holding an auditable election inside the Auditorium. The next several sections describe the voting protocol in detail; a complete list of Auditorium

messages involved can be found in Appendix A.

Bootstrapping

Each polling place will receive a number of voting machines (VOTEBox “booths,” hereafter simply referred to simply as VOTEBOXes) using the Auditorium system, as well as at least one *supervisor* machine running a special election-control application. Figure 3.7 illustrates this polling place concept.

The VOTEBOXes have identical software configurations; they differ only in their unique identifiers (assigned by the manufacturer) and in their cryptographic keys. Each public key is encapsulated into a certificate signed by a trusted certificate authority (held, for example, by the administrator in charge of elections). The corresponding private key is used for signing messages in the Auditorium protocol. Similar properties hold for supervisor machines; spares of each may be kept (in storage or active and on the network) to be brought into service at any time.

On election day, the machines are connected to power and the network and then booted. All nodes (machines and supervisor) are on the same network segment, and so can use broadcast packets to discover the presence of other nodes, self-select IP addresses, and so on. A just-started VOTEBOX uses an untrusted broadcast message to discover other nodes, and then opens a TCP connection to one or more of them in order to join the network.

The joining process is a specific case of a general fixed-point algorithm to merge two fully-connected networks. Given a node *A* wishing to join its network to the network of node *B*:

1. *A* connects to machine *B* (on a well-known port).
2. *B* sends *A* a join message, which includes its unique identifier, its public key certificate, and the set of nodes (IP addresses and identifiers) *B* knows about.
3. Satisfied (based on the signature on *B*'s certificate) that *B* is a valid voting machine, *A* sends its own join message to *B*.
4. At this point the network is now connected, but not fully. To finish connecting the network, *A* and *B* may now consult the list of nodes each received from the other, and initiate connections to any previously-unknown nodes. The bootstrapping process repeats on each such new link.

Because the set of nodes is finite, the bootstrapping process will terminate when the network settles into a fully-connected steady state. In the case of a single node A joining a network that is already fully connected, this algorithm will halt after n rounds (A will make one connection to B and then one to each of its neighbors).

join messages sent during bootstrapping are the only Auditorium messages that are not broadcast. Once a node has joined the network, it is responsible for broadcasting any new message it originates, as well as re-broadcasting messages it has never seen before, according to the simplistic gossip protocol described in Section 3.3.6. The full set of messages involved in this process can be found in Appendix A.

Opening the polls and authorizing ballots

When the election is to begin, the supervisor announces a `polls-open` message. At this point the polling place is ready to accept votes. Once a voter has signed in with poll workers, the poll workers in turn use the supervisor's user interface to authorize a particular `VOTEBOX` terminal to present the correct ballot for the voter. This is done with an `authorized-to-cast` message, which includes the ballot definition for that voter's precinct, a nonce, and the ID of the particular `VOTEBOX` the voter is to use. (Poll workers will then direct the voter to the correct machine.) All of these messages are broadcast to and recorded by every other `VOTEBOX`.

Casting and cancelling ballots

The `VOTEBOX` uses the ballot definition inside the authorization message to present a voting interface to the voter. When the voter has finished making selections and presses the final "cast ballot" button in the UI, the `VOTEBOX` announces a `cast-ballot` message, which contains the original authorization nonce for that ballot as well as an *encrypted cast ballot* containing the voter's selections. Ballot encryption is discussed further in Section 3.3.7.

Between the `authorized-to-cast` and `cast-ballot` messages, the voting machine is trusted to faithfully capture and record the voter's selections. Of course, faulty software might well tamper with or corrupt the vote before it is broadcast; see Section 3.4.3 for more on this problem and Chapter 4 for

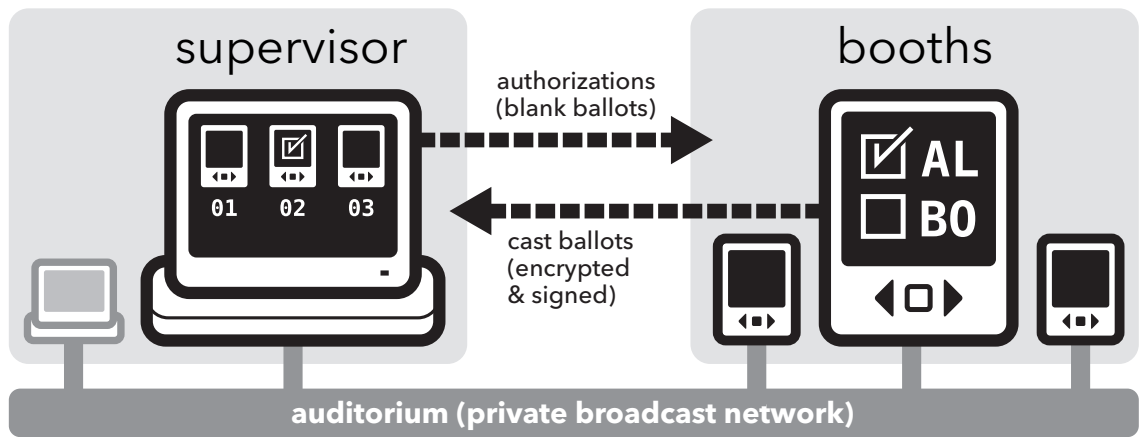


Figure 3.7: Configuration of machines in a polling place. A number of VOTEBOXes are connected to one another and to a supervisor machine by the Auditorium broadcast network. A redundant supervisor is also present on the network in case the primary supervisor fails; similarly, VOTEBOXes may be swapped out at any time.

VOTEBOX's solution.

In order to provide meaningful feedback to the voter that his ballot has been successfully cast, another message is introduced. The supervisor will acknowledge the receipt of a legitimate encrypted cast ballot by announcing a *received-ballot* message including the appropriate authorization nonce. This has several benefits: it allows the voting machine to display a confirmation message to the user; it de-authorizes the nonce, ensuring it cannot be used again to cast a legitimate ballot; and it tightly constrains the cast ballot in time by entangling the *cast-ballot* message with one from the supervisor's timeline.

If the voter flees (that is, decides to leave without casting a ballot), the appropriate procedure may be for a poll worker to cancel that outstanding ballot using the supervisor. The supervisor will then announce a *cancelled-ballot* message, which contains the nonce of the authorization to be revoked. No subsequent *cast-ballot* message corresponding to the authorization (*viz.*, including its nonce) will be considered legitimate. Likewise, the supervisor will never send an *authorized-to-cast* message for a machine with an outstanding authorization; the previous ballot must first be either cast or cancelled before a new one can be authorized.

Ballot storage and encryption

Cast ballots are part of the Auditorium event timeline, and so their order can by definition be reconstructed. This clearly poses a tradeoff with the anonymity of the voter; in general, perfect anonymity tends to stand in the way of auditing. Auditorium mitigates this particular threat to anonymity by sealing the contents of cast ballots. VOTEBOX machines use public-key cryptography to control access to the plaintext of each ballot. Each ballot is encrypted using the well-known public key for the election, distributed to all VOTEBOXes in advance. The corresponding private key is retained by the official(s) trusted to count the votes, typically administrators at the local or state level. No attacker in possession of the Auditorium logs—perhaps a network eavesdropper, or a malicious party in possession of a VOTEBOX after election day—can recover any votes, even though all votes are dutifully logged by every VOTEBOX in the polling place. Chapter 4 explores the cryptography of VOTEBOX in greater detail.

Some jurisdictions require each polling place to produce its own legible election results once the polls close, forcing the design to accommodate decryption in the polling place. This necessitates storing sensitive private key material on supervisor machines so that totals may be computed by poll workers. Precincts may also be required to produce a full paper record of each cast ballot. In this case, officials must of course be careful not to print those ballots themselves in the order they were found in the log, thereby compromising the anonymity of the votes. A straightforward way to deterministically erase the order of the ballots is to sort them lexicographically [10], although this still requires trust in the supervisor to perform the sort.

By contrast, Molnar et al. [71] use a dedicated write-only hardware device to store plaintext ballots in such a way that their order is destroyed (thus preserving anonymity) and that tampering with the finalized vote record is evident. Similarly, history-hiding append-only signatures—as proposed recently by Bethencourt et al. [12]—are software-only cryptographic functions that accomplish the same task. Unfortunately, the history-hiding properties of these ballot storage techniques conflict with the auditability goals stated here. Ballots that should not be counted (e.g., because they were cast on the wrong day) would be scrubbed of their context and become indistinguishable from legitimate votes. The Auditorium, based as it is on secure timelines, intentionally *preserves* voting history, giving

auditors enough information to establish the validity of each vote if necessary.

Heartbeats; closing the polls

It's possible that a quiescent polling place will go a long time (minutes or even hours) between legitimate election events. This will result in a loss of temporal resolution when later examining the audit logs; to address this, the supervisor will send periodic `heartbeat` messages to help fix surrounding events in wall-clock time. `VoteBoxes` also send `heartbeat` messages that include additional state, such as the battery level and the protected count. If a `VOTEBOX` observes that one of its open `TCP` connections has been reset, it announces a `disappeared` message to document the loss of connectivity to the peer at the other end of the socket. All of these messages aid auditors who might be trying to reconstruct the state of the polling place at any given time during the day.

When it is time to close the polls, the poll worker instructs the supervisor to wait until there are no further outstanding ballots before broadcasting a `polls-closed` message. This establishes an end to legitimate voting, and places each `VOTEBOX` in its post-election shutdown mode.

3.4 Robustness to failure and attack

The Auditorium design was prompted by irregularities and ambiguities encountered in a real election, as described in Section 3.2. This section catalogs the possible causes, both benign and malicious, of those problems and show how the Auditorium can detect, record, and recover from them.

3.4.1 Failures and mistakes (and attacks masquerading as such)

Early machine exit. *Scenario:* A `VOTEBOX` suddenly departs the Auditorium network. The machine may be inoperable and any storage lost. *Response:* All ballots cast from the failed machine are replicated on other `VOTEBOXes` and on the supervisor; they can be counted as part of the final tally. Note also that because `VOTEBOXes` are interchangeable, a machine may be brought in from storage or from another precinct to replace the failed machine.

Late machine entry. *Scenario:* A `VOTEBOX` enters the Auditorium after the election has started; it claims to have recorded no votes. (See Figure 3.2 for a real-life example.) *Possible causes:* (1) A

machine was brought on-line during the day to assist additional voters or to replace a failed machine.

(2) A machine was erased, possibly by accident, destroying legitimate votes cast earlier in the day.

Response: Auditors can look at the logs of other VOTEBOXes and of the supervisor to see if the machine in question cast any votes earlier in the day. Any such votes can be safely counted in the final tally, and the cause of the machine's erasure investigated after the election. Any new votes cast by the machine after joining will be recorded as normal.

Machine re-entry. *Scenario:* A VOTEBOX leaves the network and re-enters it some time later. *Possible causes:* Machine crash; temporary isolated power or network interruption. *Response:* The response is identical to that for early machine exit and late machine entry. The logs held by the supervisor and other VOTEBOXes regarding the previous activity of the re-entered machine are tamper-evident. The log on the machine in question may have missed messages during its downtime and thus may have a gap, but it will quickly re-join the global entanglement and continue to participate in voting.

Extraneous cast ballots. *Scenario:* A VOTEBOX appears to have votes cast on the wrong day. *Possible causes:* (1) Clock set wrong. (2) Test votes accidentally considered as possible real votes. (3) Intentional, malicious attempt to subvert the correct tally by stuffing the ballot box with illegitimate votes before or after the election. *Response:* For (1), if the votes are valid, their local (erroneous) timestamps are irrelevant; they will be provable successors to the polls-open event and predecessors of the polls-closed event. The situations (2) and (3), based on the definition in Section 3.3.1 of legitimate votes, will be detected when the logs are analyzed. A vote cast outside of the temporal bounds of the election (e.g. a test vote or a stuffed ballot) will have no provable link to the entangled timeline, nor will it succeed a valid **authorized-to-cast** message, so it will be considered illegal.

Electrical failure. *Scenario:* Electricity fails at the polling place. *Response:* This is a known problem with electronic voting machines of any sort. Most modern DRES have battery backups; there is no reason the supervisor and network switch can not also be backed up in this way (perhaps as simply as plugging them into a UPS). Battery status is part of the heartbeat message, so the supervisor can display the status of each VOTEBOX.

3.4.2 Mega attacks

I now present a class of possible (if implausible) threats to election integrity, which I refer to as “mega attacks.” These require either widespread collusion or overwhelming force in order to execute, but the risk—attackers able to exert total control over the outcome of an election—is just as extreme. Most voting systems, electronic or otherwise, are vulnerable to such attacks; the goal for voting in the Auditorium is to be able to recover from these attacks where possible, and to detect them in any case.

Switched results. *Scenario:* Parties in possession of all election equipment the night before the election (a so-called “sleepover”) use the hardware to conduct a secret election with a chosen outcome. On election day, voters cast ballots as normal, but the attackers substitute the results of the secret election (including cast ballots and entangled logs) when the polls are closed. *Response:* This attack is equally effective against paper ballots, and should be addressed with human procedures. For example, no single party should be allowed unsupervised custody of the machines to be used on election day. Alternatively, distribution of supervisors should be delayed until election morning.

The properties of the Auditorium allow this attack to be detected. On the morning of the election, the election administrator can distribute to each polling place a nonce to be input into each supervisor. This nonce, hard to guess but easy to input, might take the form of a few English words. The supervisor would require the user to input the nonce before opening the polls; the nonce would then be embedded in the polls-open message. Any audit can examine the polls-open message to see if the nonce is correct (modulo minor keying errors by the poll workers). A sleepover conspiracy would be unable to guess the correct nonce to inject into their polls-open message, and thus the Auditorium record of their secret election will be detectably invalid.

A similar defense will thwart attackers who run a secret election and switch the results *after* election day. This is accomplished by immediately publishing the hash value of the polls-closed message. Copies may go to election observers, newspapers, and so forth. This effectively seals the results of the legitimate election, making it impossible to substitute new results later.

Shadow election. *Scenario:* Similar to the switched-results attack, a conspiracy of election workers substitutes false election results for the real ones. Instead of conducting the election the night

before, they conduct the secret election on election day as a “shadow” of the real one, so they have access to any “launch code” nonces used to validate the date and time of the election. At the end of the election, they report the polls-closed message from their shadow election. *Response:* In order to conduct a shadow election, the attackers will need to duplicate the entire voting apparatus, down to the private keys used by each VOTEBOX to sign messages. (Creating new keypairs won’t work; the correct keys are enclosed in certificates signed by election officials.) Making this duplication difficult requires hardware-based protection schemes, such as trusted platform module (TPM) chips which resist extraction of key material.

Booth capture. *Scenario:* Armed attackers take control of the polling place by force and stuff ballots until the police arrive. *Response:* Western readers may find this attack implausible, but such events are not uncommon in fledgling democracies and have occurred in India as recently as 2004 [81]. Attackers have two potential goals: (1) cast fraudulent ballots; (2) destroy legitimate ballots. The danger of (1) is mitigated by estimating when the attack took place (perhaps by allowing poll workers to broadcast an “election compromised” Auditorium message, rather like an alarm button at a bank) and discarding votes cast after that point.

In the case of (2), auditors can recover votes from any machine not destroyed, but they cannot recover from complete destruction of the polling place. The only defense would be a network link to an offsite location (over which Auditorium messages would be broadcast, just as within the polling place). Removing the air-gap between the precinct network and the “outside world,” would greatly increase the attack surface of the polling place, thereby introducing unacceptable risk.

In either case, this attack is trivially detectable, if not always recoverable.

3.4.3 Software tampering

Finally I consider software tampering, a critical issue with any form of electronic voting. While the Auditorium focuses on issues of auditability arising when correct voting systems fail or are used incorrectly, any reasonable threat model for electronic voting includes the introduction of malicious code into the overall system.

The design of Auditorium makes it very hard for a malicious voting machine to corrupt the en-

tangled record once it has been committed. Hash chaining prevents insertion of spurious events; digital signatures prevent forgery. Digital signatures also allow VOTEBOXes to unambiguously identify the source of each message, so a node wishing to deny service by, for example, filling up the disks of its peers with junk messages can be effectively ignored by other nodes. As shown in Section 3.3.6, the gossip protocol of the Auditorium makes it impossible for a malicious node to maintain multiple divergent timelines without being detected. Any of the above events or other unusual activity, if found in the audit logs, will necessitate impounding and further investigation of the equipment used in the election.

Malicious software on a single VOTEBOX would not be able to cast unattended votes without a corrupt supervisor, as only the supervisor can generate the necessary `authorized-to-cast` message. However, the VOTEBOX could easily show the voter his or her correct selection while quietly casting a vote for somebody else. This is the *cast as intended* property, identified in Chapter 2 as extremely hard for a DRE to verifiably satisfy. Addressing this fundamental problem requires mechanisms beyond the Auditorium; such techniques are the subject of the following chapter.

CHAPTER 4

VERIFIABILITY: THE CAST-AS-INTENDED CHALLENGE

4.1 Introduction

With Auditorium, auditors of a contested election have a powerful tool that provides conclusive evidence about the legitimacy of a particular ballot. The assumption, however, is that that ballot—when introduced into the Auditorium network and logs by a VOTEBOX under the control of a voter—is initially correct. While it is essential to defend against tampering with (or destroying) election records, a DRE must be trusted to faithfully capture the voter’s intent when creating those records in the first place.

This is the heart of the *cast as intended* property, introduced in Chapter 2, required of any voting system that is to be considered verifiable. VOTEBOX must somehow prove to the voter that the ballot recorded for counting is the ballot created and reviewed for correctness by the voter.

In this chapter I describe a novel application of a cryptographic approach to voting pioneered by Benaloh [11]. This technique allows VOTEBOX to satisfy this stringent verifiability property; when used in concert with Auditorium, the technique becomes more practical for those wishing to verify a VOTEBOX. The cryptography used here also offers greater protection for the voter’s privacy than the Auditorium system can provide on its own.

4.2 Voter privacy; encryption

The privacy (or, more specifically, *anonymity*) of a voter is an essential ingredient in any voting system; without privacy, a voter can be coerced or bribed to vote in some particular way. A voting system that safeguards the voter's anonymity prevents him from proving how he voted to anyone (or, put another way, allows him to claim with equal believability any particular vote).

The Auditorium log does not obviously violate this anonymity property: nowhere in the record is the voter's name or any other identifying information associated with each ballot. However, by design it contains a strong proof of *order* of election events, and this includes the ballots themselves. This ordering can in fact be used to expose the identity of the voter. An attacker with access to chronologically-ordered lists of both the ballots and the voters in a particular polling place can establish with some probability a mapping from voter to ballot.

Two classes of approaches exist to prevent this compromise of voter anonymity:

1. The order of the ballots may be destroyed, for example, by shuffling them.¹
2. The contents of the ballots may be concealed, for example, by encrypting them.

Shuffling ballots is problematic because the “shuffled” order may not be random at all; a malicious voting machine could choose a null shuffle (that is, preserving exactly the original order) or could even encode information in the particular shuffle it chooses. It has been noted by Benaloh [10] that lexicographic ordering of ballots is an effective way to destroy order without opening these covert channels. Because there is only one sorted order, a voting machine is not free to choose a convenient ordering.

In the context of VOTEBOX, however, ballot re-ordering presents another problem. If ballot plaintexts are excised from the (provably-ordered) Auditorium log, their validity (per the requirements of Section 3.3.1) can no longer be conclusively established. A two-part auditing record—one that contains the order of ballots but not their contents, and a second that contains the contents of ballots in an arbitrary non-chronological order—might allow an auditor to discover *whether* any ballots ex-

¹Note that we cannot similarly destroy the order of the voters; an attacker can wait outside a polling place, recording in order the identities of voters as they leave.

ist in the record that should not be counted (because they occurred outside the time bounds of the election, for example), but not *which ballots* should be disqualified.

This brings us to the second technique: ballot encryption. By including in the Auditorium log the *ciphertext* of a ballot, we prevent anyone with access to that log from matching ballot *plaintexts* with voters. It is natural to choose a public-key cryptosystem here, as there are many writers of sensitive information (each VOTEBOX, creating encrypted ballots) and few readers (the elections officials responsible for counting the ballots). A straightforward implementation would use a single keypair, retaining the private portion for officials and distributing the public key to every voting machine.

4.3 Tallying encrypted ballots

There is, however, a problem with this approach. Counting ballots naturally requires that they be decrypted. This means that the elections officials will at some point during the tallying process see the decrypted ballots in order. It also means that jurisdictions that legally require each precinct to immediately post the tally of all votes cast in its polling place can no longer do so; alternatively, such precincts must each have a copy of the private decryption key, dramatically widening the circle of trusted parties.

Mixnet-based cryptographic voting systems [21, 25, 5, 27] address this problem by combining the re-ordering and encryption techniques described in the previous section. Each stage of a verifiable re-encryption mixnet reorders and partially decrypts its inputs in such a way as to prove that the output set is equivalent (although its order is randomized). Ballots, which enter the mix in order but encrypted, exit in cleartext but also in some other order; this allows them to be tallied without exposing the original order (and hence the voter's identity).

VOTEBOX uses a different approach, called *homomorphic encryption*, to solve the problem of tallying votes without decrypting them in order. This choice is partially to avoid some of the technical problems that plague mixnet designs (including establishing and coordinating the stages of the mixnet among multiple parties); moreover, the homomorphic scheme permits application of a verification mechanism proposed by Benaloh [11] that allows voters to verify the correct operation of VOTEBOX on election day.

4.3.1 Ballots as counter vectors

We begin by encoding a cast ballot as an n -tuple of integers, each of which can be 1 or 0. Each element of the n -tuple represents a single choice a voter can make, n is the number of choices, and a value of 1 encodes a vote *for* the choice while 0 encodes a vote *against* the choice. (In the case of propositions, both “yes” and “no” each appear as a single “choice,” and in the case of candidates, each candidate is a single “choice.”) The cast ballot structure needs not be organized into races or contests; it is simply an opaque list of choice values. We define each element as an integer (rather than a bit) so that ballots can be homomorphically combined. That is, ballots $A = (a_0, a_1, \dots)$ and $B = (b_0, b_1, \dots)$ can be summed together to produce a third ballot $S = (a_0 + b_0, a_1 + b_1, \dots)$, whose elements are the total number of votes for each choice.²

4.3.2 Homomorphic encryption of counters

VOTEBOX uses an El Gamal variant that is additively homomorphic to encrypt ballots before they are cast. Each element of the tuple is independently encrypted. The encryption and decryption functions are defined as follows:

$$E(c, r, g^a) = \langle g^r, (g^a)^r f^c \rangle \quad (4.1)$$

$$D(\langle g^r, g^{ar} f^c \rangle, a) = \frac{g^{ar} f^c}{(g^r)^a} \quad (4.2)$$

$$D(\langle g^r, g^{ar} f^c \rangle, r) = \frac{g^{ar} f^c}{(g^a)^r} \quad (4.3)$$

where f and g are group generators, c is the plaintext counter, r is randomly generated at encryption time, a is the decryption key, and g^a is the public encryption key. To decrypt, a party needs either a or r in order to construct g^{ar} . (g^r , which is given as the first element of the cipher tuple, can be raised to a , or g^a , which is the public encryption key, can be raised to r .) After constructing g^{ar} , the decrypting party should divide the second element of the cipher tuple by this value, resulting in f^c .

To recover the counter’s actual value c , we must invert the discrete logarithm f^c , which of course

²While this simple counter-based ballot does not accommodate write-in votes, homomorphic schemes exist that allow more flexible ballot designs, including write-ins [65].

is difficult. As is conventional in such a situation, we accelerate this task by precomputing a reverse mapping of $f^x \rightarrow x$ for $0 < x \leq M$ (for some large M) so that for expected integral values of c the search takes constant time. (We fall back to a linear search, starting at $M + 1$, if c is not in the table.)

We now show that our encryption function is additively homomorphic by showing that when two ciphers are multiplied, their corresponding counters are added:

$$E(c_1, r_1) \odot E(c_2, r_2) = \langle g^{r_1}, g^{ar_1} f^{c_1} \rangle \odot \langle g^{r_2}, g^{ar_2} f^{c_2} \rangle \quad (4.4)$$

$$= \langle g^{r_1+r_2}, g^{a(r_1+r_2)} f^{c_1+c_2} \rangle \quad (4.5)$$

This cryptosystem allows an election's results to be computed by multiplying ciphertexts to arrive at an encrypted tally, then decrypting it. (Subtleties that must be addressed to maintain the security of the approach are described in Section 4.6.) In the following section I show how this cryptosystem may also be used to arrive at a powerful result: the elusive *cast as intended* property of verifiable voting systems.

4.4 Verifiability through ballot challenge

4.4.1 Immediate ballot challenge

To allow the voter to verify that her ballot was cast as intended, we need some way to prove to the voter that the encrypted ballot published in the Auditorium log represents the choices she actually made. After all, a DRE can by definition alter the contents of a ballot at any time, particularly before that ballot is ever entered in a secure log or broadcast over Auditorium (which would in this case faithfully replicate the wrong information).

VOTEBOX's solution to the cast-as-intended problem, here termed "immediate ballot challenge," an adaptation of a technique due originally to Benaloh [11]. It is a probabilistic voter-initiated auditing technique, allowing any voter to challenge the voting machine to prove that it is casting ballots faithfully and correctly. Of course, because these challenges generally force the voting machine to reveal information that would compromise the anonymity of the voter, challenged ballots must be discarded and not counted in the election. Note that by using what appears in every way to be a real

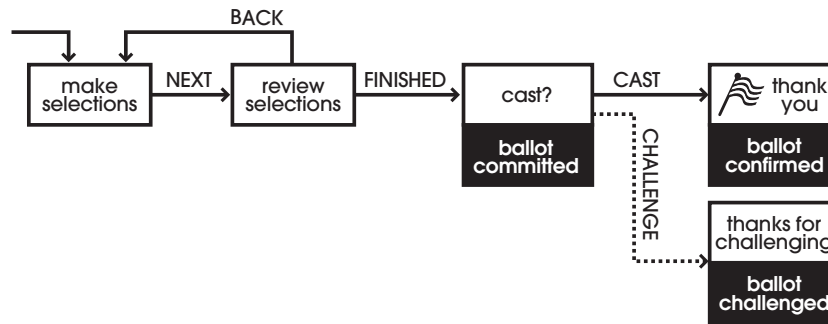


Figure 4.1: Challenge flow chart. As the voter advances past the review screen to the final confirmation screen, VOTEBOX commits to the state of the ballot by encrypting and publishing it. A challenger, having received this commitment (the encrypted ballot) out-of-band (see Figure 4.2), can now invoke the “challenge” function on the VOTEBOX, compelling it to reveal the contents of the same encrypted ballot. (A voter will instead simply choose “cast”.)

ballot for this challenge, the auditor avoids artificial testing conditions (such as are common in “logic and accuracy tests”) that can usually be detected by a voting machine. With the challenge system, a malicious voting system now has no knowledge of which ballots will be challenged, so it must either cast them all correctly or risk being caught if it misbehaves.

Our implementation of this idea is as follows. Before a voter has committed to her vote, in most systems, she is presented with a final confirmation page which offers two options: (1) go back and change selections, or (2) commit the vote. Our system, like Benaloh’s, adds one more page at the end, giving the voter the opportunity to challenge or cast a vote. At this point, Benaloh prints a paper commitment to the vote. VOTEBOX will similarly encrypt and publish the cast ballot *before* displaying this final “challenge or cast” screen. If the voter chooses to cast her vote, VOTEBOX simply logs this choice and behaves as one would expect, but if the voter, instead, chooses to *challenge* VOTEBOX, it will publish the value for r that it passed to the encryption function (defined in Equation 4.1) when it encrypted the ballot in question. Using Equation 4.3 and this provided value of r , any party (including the voter) can decrypt and verify the contents of the ballot without knowing the decryption key. An illustration of this sequence of events is in Figure 4.1.

In order to make this process *immediate*, we need a way for voters (or voter advocates) to safely observe Auditorium traffic and capture their own copy of the log. It is only then that the voter will

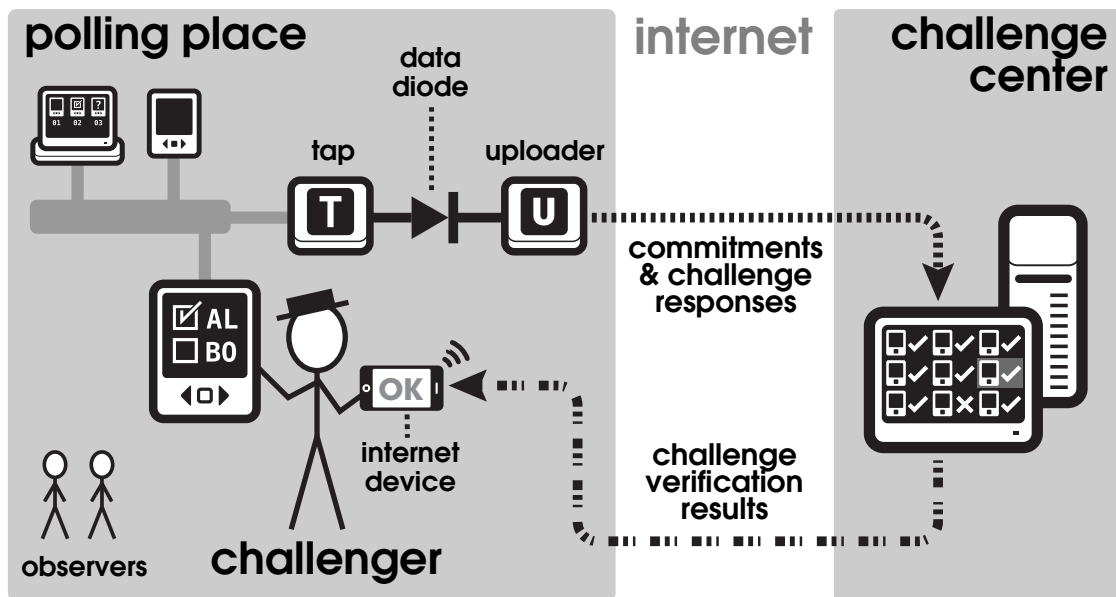


Figure 4.2: Voting with ballot challenges. The VOTEBOX polling place sends a copy of all Auditorium log data over a one-way channel to election headquarters (not shown) which aggregates this data from many different precincts and republishes it. This enables third-party “challenge centers” to provide challenge verification services to the field.

be able to check, in real time, that VOTEBOX recorded and encrypted her preferences correctly. To do this, we propose that the local network constructed at the polling place be connected to the public Internet via a data diode [61], a physical device which will guarantee that the information flow is one way.³ This connectivity will allow any interested party to watch the polling location’s Auditorium traffic in real time. In fact, any party could provide a web interface, suitable for access via smart phones, that could be used to see the voting challenges and perform the necessary cryptography. This arrangement is summarized in Figure 4.2. Additionally, on the output side of the data diode, we could provide a standard Ethernet hub, allowing challengers to locally plug in their own auditing equipment without relying on the election authority’s network infrastructure. Because all Auditorium messages are digitally signed, there is no risk of the challenger being able to forge these messages.

³An interesting risk with a data diode is ensuring that it is installed properly. Polling place systems could attempt to ping known Internet hosts or otherwise map the local network topology, complaining if two-way connectivity can be established. We could also imagine color-coding cables and plugs to clarify how they must be connected.

4.4.2 Implications of the challenge scheme

Many states have laws against connecting voting machines or tabulation equipment to the Internet—a good idea, given the known security flaws in present equipment. Our cryptographic techniques, combined with the data diode to preserve data within the precinct, offer some mitigation against the risks of corruption in the tallying infrastructure. An observer could certainly measure the voting volume of every precinct in real-time. This is not generally considered to be private information.

VOTEBOX systems do not need a printer on every voting machine; however, Benaloh’s printed ballot commitments offer one possibly valuable benefit: they allow any voter to take the printout home, punch the serial number into a web site, and verify the specific ballot ciphertext that belongs to them is part of the final tally, thus improving voters’ confidence that their votes were counted as cast. A VOTEBOX lacking this printer cannot offer voters this opportunity to verify the presence of their own cast ballot ciphertexts. Challengers, of course, can verify that the ciphertexts are correctly encrypted and present in the log in real-time, thus increasing the confidence of normal voters that their votes are likewise present to be counted as cast. Optionally, Benaloh’s printer mechanism could be added to VOTEBOX, allowing voters to take home a printed receipt specifying the ciphertext of their ballot.

Similarly, VOTEBOX systems do not need NIZKS. While NIZKS impose limits on the extent to which a malicious VOTEBOX can corrupt the election tallies by corrupting individual votes, this sort of misbehavior can be detected through our challenge mechanism. Regardless, NIZKS would integrate easily with our system and would provide an important “sanity checking” function that can apply to *every* ballot, rather than only the challenged ballots.

4.5 Procedures: administering a VOTEBOX election

To summarize the ballot challenge design, let us review the steps involved in conducting an election with the system.

4.5.1 Before the election

1. The ballot preparation software is used to create the necessary ballot definitions.
2. Ballot definitions are independently reviewed for correctness (so that the ballot preparation software need not be trusted).
3. Ballot definitions and key material (for vote encryption) are distributed to polling places along with VOTEBOX equipment.

4.5.2 Opening the polls

4. The Auditorium network is established and connected to the outside world through a data diode.
5. All supervisor consoles are powered on, connected to the Auditorium network, and one of them is enabled as the primary console (others are present for failover purposes).
6. Booth machines are powered on and connected to the Auditorium network.
7. A “launch code” is distributed to the polling place by the election administrator.
8. Poll workers open the polls by entering the launch code.

The last step results in a “polls-open” Auditorium message, which includes the launch code. All subsequent events that occur will, by virtue of hash chaining, provably have occurred *after* this “polls-open” message, which in turn means they will have provably occurred on or after election day.

4.5.3 Casting votes

9. The poll worker interacts with the supervisor console to enable a booth for the voter to use. This includes selecting a machine designated as not in use and pressing an “authorize” button.
10. The supervisor console broadcasts an authorization message directing the selected machine to interact with a voter, capture his preference, and broadcast back the result.
11. If the booth does not have a copy of the ballot definition mentioned in the authorization message, it requests that the supervisor console publish the ballot definition in a broadcast.
12. The booth graphically presents the ballot to the voter and interacts with her, capturing her

choices.

13. The booth shows a review screen, listing the voter's choices.
14. If the voter needs to make changes, she can do that by navigating backward through the ballot screens. Otherwise, she indicates she is satisfied with her selections.
15. The booth publishes the encrypted ballot over the network, thereby committing to its contents. The voter may now choose one of two paths to complete her voting session:

Cast her vote by pressing a physical button. The VOTEBOX signals to the voter that she may exit the booth area; it also publishes a message declaring that the encrypted ballot has been officially cast and can no longer be challenged.

Challenge the machine by invoking a separate UI function. The challenged VOTEBOX must now reveal proof that the ballot was cast correctly. It does so by publishing the secret r used to encrypt the ballot; the ballot is no longer secret. This proof, like all Auditorium traffic, is relayed to the outside world, where a challenge verifier can validate against the earlier commitment and determine whether the machine was behaving correctly. The voter or poll workers can contact the challenge verifier out-of-band (e.g., with a smartphone's web browser) to discover the result of this challenge. Finally, the ballot committed to in step 15 is nullified by the existence of the proof in the log. The VOTEBOX resets its state. The challenge is complete.

4.5.4 Closing the polls

16. A poll worker interacts with the supervisor console, instructing it to close the polls.
17. The supervisor console broadcasts a "polls-closed" message, which is the final message that needs to go in the global log. The hash of this message is summarized on the supervisor console.
18. Poll workers note this value and promptly distribute it outside the polling place, fixing the end of the election in time (just as the beginning was fixed by the launch code).
19. Poll workers are now free to disconnect and power off VOTEBOXes.

4.6 Security of the cryptosystem and ballot challenge technique

4.6.1 Ballot decryption key material

We have thus far avoided the topic of which parties are entitled to decrypt the finished tally, assuming that there exists a single entity (perhaps the director of elections) holding an El Gamal private key. We can instead break the decryption key up into shares [95, 30] and distribute them to several mutually-untrusting individuals, such as representatives of each major political party, forcing them to cooperate to view the final totals.

This may be insufficient to accommodate varying legal requirements. Some jurisdictions require that each county, or even each polling place, be able to generate its own tallies on the spot once the polls close. In this case we must create separate key material for each tallying party, complicating the matter of who should hold the decryption key. Our design frees us to place the decryption key on, e.g., the supervisor console, or a USB key held by a local election administrator. We can also use threshold decryption to distribute key shares among multiple VOTEBOXes in the polling place or among mutually-untrusting individuals present in the polling place.

4.6.2 Covert channels in randomized ciphers

This El Gamal-based cryptosystem, like many others, relies on the generation of random numbers as part of the encryption process. Since the ciphertext includes g^r , a malicious voting machine could perform $O(2^k)$ computations to encode k bits in g^r , perhaps leaking information about voters' selections. Karlof et al. [62] suggest several possible solutions, including the use of trusted hardware. Verifiable randomness may also be possible as a network service or a multi-party computation within the VOTEBOX network [40].

4.7 Attacks on the challenge system

A subtle source of attacks on such cryptographic measures, proposed by Kelsey et al. [63], comes in the form of covert communication channels between the voting machine and the voter. This may seem counter-intuitive: is the voting machine not *supposed* to communicate with the voter? Yet, a

malicious voting machine could cooperate with some present human to subvert the challenge system. For instance, if a poll worker could somehow communicate to the VOTEBOX (e.g. through a “secret knock” of predetermined, seemingly-innocuous UI actions) that the machine was about to be challenged by a voter, the machine could easily escape detection by operating correctly for that voting session.

The voter, acting as a challenger, may also collude with the machine. A malicious VOTEBOX could send a signal to the voter, indicating whether or not she should challenge, using something as unobtrusive as the parity of the vote’s ciphertext (by subverting the randomness of the cipher as in Section 4.6.2). An external observer could then catch her if she failed to vote as intended. Benaloh solves this particular problem by having the paper commitment hidden behind an opaque shield; she can see that the commitment has been made, but nothing more. However, one can envision very subtle cues in the voting UI that have the same effect (a color change, typographic error, etc.); the entire voting machine cannot be hidden behind an opaque shield!

To address attacks of this form, we must constrain a would-be challenger’s behavior. A voter must declare to poll workers, after being authorized to vote (i.e., after the `authorized-to-cast` message) but before entering the booth, that she intends to challenge the VOTEBOX. At this point, a poll worker should supervise the challenge while the challenger proceeds to vote. While the VOTEBOX has no idea it is being challenged, the voter (or absolutely anybody else) can freely use the machine, videotape the screen, and observe its network behavior. The challenger must not, however, be allowed to cast the ballot. (There might even be a physical cover that can be locked on the outside of a hardware “cast ballot” button enforcing this.)

Similarly, undeclared challenges must not be allowed. Recall that because a challenged ballot is exposed to the network (and, consequently, the world) by revealing the one-time decryption value r , such ballots are *spoiled* and must never be tallied. By showing the user a cast ballot confirmation screen while secretly issuing challenges, the voting machine can disenfranchise any voter or defraud any candidate. To address these phantom challenges, we take advantage of Auditorium. Challenge messages, which are broadcast to the entire network, initiate a suitable alarm on the supervisor console. For a genuine challenge, the supervisor will be expecting the alarm. Otherwise, the unexpected

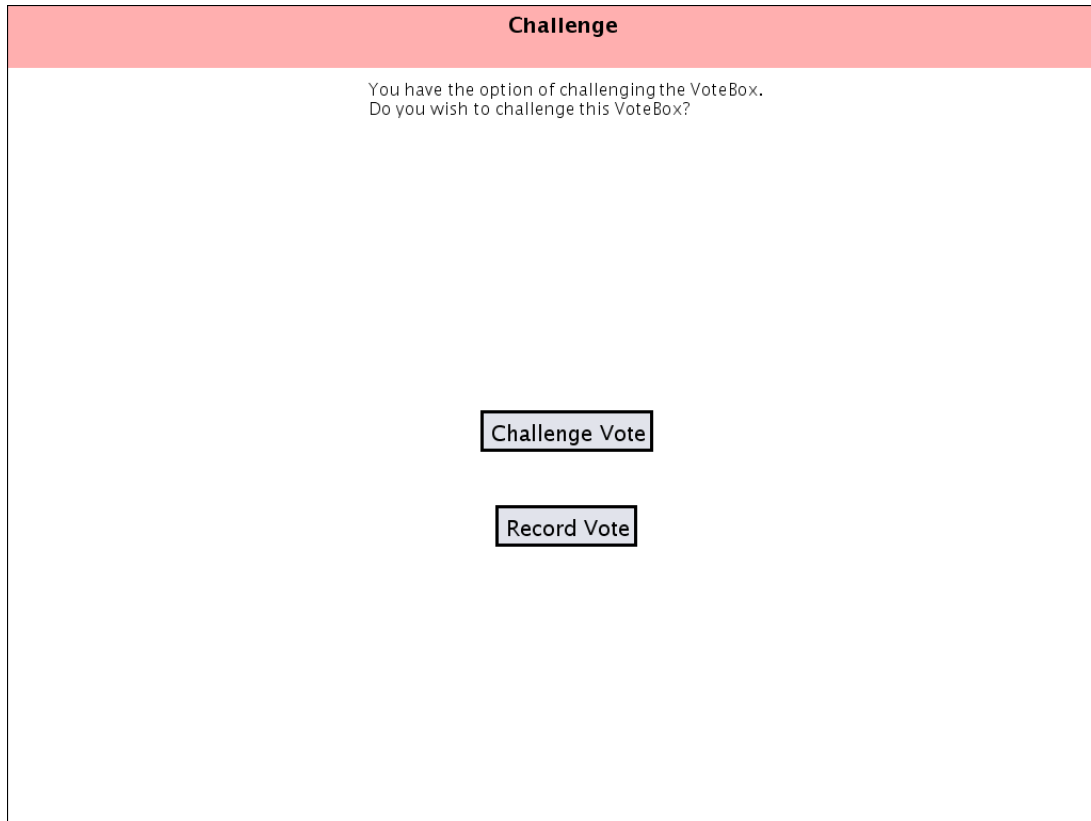


Figure 4.3: Final VOTEBOX screen: challenge, or record? This decision point comes after the voter has already confirmed her choices on the review screen; note that she has no way to go “back” to modify the ballot at this point, as her ballot has already been encrypted and broadcast in the Auditorium as part of the VOTEBOX commitment. Compare with the “Previous Page” and “Next Page” buttons visible in Figure 6.1, representative of every other screen in the VOTEBOX ballot. Note also that this user experience is entirely determined by the graphics and layout logic in the current ballot creation tool (Section 6.2); the VOTEBOX booth software is agnostic to the presentation of these functions, including graphic treatment and even placement in the overall flow of the ballot.

alarm would cue a supervisor to offer the voter a chance to vote again (perhaps on another VOTEBOX, as the one that issued the spurious challenge may need to be removed from service for examination).

There is a related user interface issue here: How do we present the option to challenge a ballot in such a way that a voter who does not wish to use the challenge feature (and indeed may not be aware of its existence) cannot accidentally challenge her ballot? This question is open. In VOTEBOX, the “challenge” button is clearly visible (Figure 4.3), on par with the “cast” button (for ease of development and demonstration). In practice, the challenge button should be more unobtrusive to

deter or eliminate accidental invocation. Still, some number of non-challenging voters will accidentally challenge their ballots. This problem is mitigated by the same measure used to combat spurious machine-initiated challenges. Poll workers, upon observing a supervisor console alarm indicating a challenge, can explain the situation to the voter and offer them the opportunity to vote over again. Therefore, accidental challenges will only inconvenience, rather than disenfranchise, a voter.

4.8 Verifying the tabulation

The techniques described in this chapter yield a powerful result: the ability to audit VOTEBOX machines for correctness, preserving privacy, without artificial logic & accuracy test conditions. The *cast as intended* verifiability property, normally very difficult for a DRE to satisfy, is satisfied here.

What, then, of *counted as cast*? Because the Auditorium logs represent a complete and accurate account of election day events, the records may be recounted at any time using different software, provided that the recounter possesses the appropriate private key shares. This may not be enough to satisfy a critical third party who, while not entrusted with such key material, might legitimately wish to confirm that the decryption and summation are performed correctly. Chapter 2 describes a number of cryptographic voting systems that use mixnets to achieve this property, and this technique is also applicable to VOTEBOX.

A simpler approach, leveraging the homomorphic tallying used in VOTEBOX, can be used to prove to a third party that a value is decrypted faithfully.

The scenario is as follows: a third party, perhaps an entity already serving as an offsite challenge center, wishes also to verify the correctness of the decrypted tally. (In the rest of this section, “challenger” will refer to such a third party, and “prover” will refer to the state.) There are two steps: (1) independently confirming the encrypted tally, and (2) verifying that the decryption matches the encryption.

The challenger accomplishes the first step by collecting all the ciphertexts for all valid ballots (that is, those not spoiled by challenging) and multiplying them. The result is the encipherment of the overall tally; this number can be compared with a published result from election officials. If these values are not equal, the set of encrypted ballots seen by the challenge center is different from the set

seen by elections officials and can alert all parties involved to the discrepancy (which may require legal means to resolve).

Having established that the summing was performed accurately, the challenger now verifies the correctness of the decryption. This technique applies the Chaum-Pedersen protocol for proving the equivalence of discrete logarithms [24], inspired by a similar application in Helios [5]. Given the public key g^a and a ciphertext $(\alpha, \beta) = (g^r, g^{ar}m)$ of plaintext m (as given in Equation 4.1):

1. The prover chooses a random w and provides $(A, B) = (g^w, g^{rw})$.
2. The challenger sends a challenge value c to the prover.
3. The prover computes $t = w + ac$ and sends t to the challenger.
4. The challenger can now confirm that **(i)** $g^t = Ag^{ac}$ and **(ii)** $g^{rt} = B(\beta/m)^c$:

$$\begin{aligned}
 \text{(i)} \quad g^t &= g^{w+ac} \\
 &= g^w g^{ac} \\
 &= A(g^a)^c
 \end{aligned}$$

$$\begin{aligned}
 \text{(ii)} \quad g^{rt} &= g^{r(w+ac)} \\
 &= g^{rw} g^{rac} \\
 &= Bg^{rac} \\
 &= B(g^{ar})^c \\
 &= B \left(\frac{g^{ar}m}{m} \right)^c \\
 &= B(\beta/m)^c
 \end{aligned}$$

This is an interactive proof, requiring two rounds of communication between the (honest) prover and challenger. By applying the Fiat-Shamir heuristic [37], which turns interactive proofs into non-interactive ones by enforcing the challenge value c to be the output of a one-way function of the proof inputs (A, B) , this can be made non-interactive.

CHAPTER 5

QUERIFIER: SECURE LOG ANALYSIS

5.1 Integrity checking in secure logs

In this chapter I consider the problem of what to do with the secure logs of the sort meticulously captured by VOTEBOX and gossiped in the Auditorium. According to Schneier and Kelsey:

Audit logs are useless unless someone reads them. Hence, we first assume that there is a software program whose job it is to scan all audit logs and look for suspicious entries. ... After that, the details are completely dependent on the particular log entries. [92]

Current research in secure and entangled logging is content to leave the discussion here, having successfully argued that buried within such a secure log is proof of its order and integrity. But how might one go about unearthing this proof? For a log with many entries, how can an auditor locate, for example, a hole in the record left by the omission of an incriminating statement?

This low-level idea of validity—namely, that a secure log is tamper-free if its hash chains are valid and complete—can be verified mechanically. It is easy to envision a program that combs such a log, computing message digests and validating signatures. The program will likely differ for each application, owing to the idiosyncrasies of log formats, but will perform the same essential computations.

5.1.1 Application-specific properties

It is reasonable to assume that an application which makes use of such logs might want to enforce more sophisticated constraints on the behavior of its participants (as evidenced by entries in the log). For example:

- **In a banking system:** One can ask questions about transactions and balances. For example, did Alice receive a certain amount of money before spending it? This could help her prove that she never went below a minimum account balance.
- **In a stock trading system:** Ordering issues are critical to the fair execution of stock trades. For example, the SEC has charged several stock traders with eavesdropping on large institutional stock orders and trading ahead of them at better prices [94]. This evidence naturally extends to more typical communications (e.g., email, instant messaging), permitting someone accused of insider trading to prove that they issued a stock order before they acquired any insider information.
- **In a multiplayer game:** Modern online games typically rely on correct behavior of client software, and are therefore vulnerable to cheating by modified clients. Such a constraint might involve a player completing some action (e.g., a payment) before being allowed to begin another action (e.g., accessing a certain area). Violation of this constraint indicates buggy or malicious software.
- **In a file server:** Yumerefendi et al. [106] describe a networked storage service that uses secure logs as evidence when the correctness of the server is challenged. Clients “trust but verify” the server, periodically requesting from it proofs of correct behavior. Clients must check these proofs for *semantic* correctness: beyond merely being well-formed and authentic, they must conform to the operational semantics of a file server.

The motivating example in this thesis, of course, is VOTEBOX, and the many crucial election events (such as *polls opened* and *ballot cast*) that, when encoded in ordered secure logs, tell a believable tale of election day. In order to make this claim, auditors must be able to automatically examine these logs not just for breaks in the hash chain, but for violation of the “rules” of a valid election in this system. For example, each ballot to be tallied must first be shown to have been *authorized* to be cast. As described in Chapter 3, these authorizations appear in the log as *authorized-to-cast* messages originated by the supervisor when it assigns a new voter to a specific booth; one such message must therefore precede every single *cast-ballot* message.

Clearly, this rule—asserting an invariant in the secure log—makes sense only in the context of the electronic voting problem, and is in fact intimately tied to this particular system design. It is reasonable to assume that other systems that employ secure logs will have their own particular constraints, above and beyond state replication, that must be mechanically verified.

Ideally, it ought to be possible to develop a single log analysis tool that, given a log, can reach any such conclusion that an application might need. The tool should need no intrinsic understanding of the structure of the log, nor of the application's needs; nor should it achieve this generality by *imposing* those things on an application. Rather, applications should be able to *specify* what behavior is considered correct, and to do so in terms of their own log entry structure and format.

Finally, this verification tool must not rely on deterministic or repeatable behavior on the part of nonetheless correct applications. The correctness properties that are of interest are not limited to matters of faithfully replicating some piece of state across a system. That is, a system free of Byzantine faults (e.g., a buggy implementation, or a correct system that is incorrectly operated) may still violate a rule that has been established for the system. Therefore, techniques such as PBFT [19] or the recently proposed PeerReview [43], which relies on deterministic replay to identify failures, are insufficient to the task described here.

5.2 A language for log properties

I begin by proposing an abstract language for an application to articulate its constraints and queries over logs, including secure logs, to some general-purpose rule evaluator. This language is based on the well-understood constructs of first-order predicate logic. Useful properties of logs are naturally expressible in such a system. In this section, I detail the semantics of this language; discussion of how to interpret this language for concrete inputs is the subject of the following section.

5.2.1 Domain of expression

In practice, a log is a finite set of entries. An entry might take the form of a single datum, such as a character string; a list or array of strings or other data; or a more general recursive structure, possibly including other entries. The language must generalize over all such logs.

I therefore propose the following *domain* for expressions in this language. The abstract set of all *tuples* (denoted \mathbf{T}^*) has as elements all tuples whose entries are either other tuples or elements of the set of *characters* (denoted \mathbf{T}). (As a shorthand, *strings*—flat tuples of only characters—will be written in “quotes”.) In addition, \mathbf{T}^* contains the *null tuple* (ε) and the *wildcard tuple* (?).

Finally, \mathbf{T}^* is an infinite set, so define $L \subset \mathbf{T}^*$ is defined to be the finite corpus of *log entries under scrutiny*. L will refer to a set of concrete log entries that have been produced by an application and that may therefore be considered by logical rules.

5.2.2 Relations

The following relations (functions over tuple space) are defined:

Equality ($\mathbf{T}^* \times \mathbf{T}^* \rightarrow \{\text{TRUE}, \text{FALSE}\}$) $T_1 = T_2$ is defined to be true if both are ε , or if they have the same length and are recursively equal: that is, the i^{th} sub-element of T_1 and T_2 are equal for all $1 \leq i \leq |T_1|$.

Pattern matching ($\mathbf{T}^* \times \mathbf{T}^* \rightarrow \mathbf{T}^*$) A tuple T_1 *matches* the tuple pattern P (written $T_1.P$) if they are recursively equal as described above, with the following relaxation: any member of \mathbf{T}^* is also considered to be a match for special *wildcard* tuple. This permits recursive pattern-matching of tuples. For example, the tuples $\langle \text{"All"}, \text{"Men"}, \text{"Are"}, \text{"Mortal"} \rangle$ and $\langle \text{"All"}, \text{"Cats"}, \text{"Are"}, \text{"Animals"} \rangle$ both match the pattern $\langle \text{"All"}, ?, \text{"Are"}, ? \rangle$.

The result value of $T_1.P$ is defined in the affirmative case to be a tuple containing those sub-elements of T_1 that correspond to the wildcards in the pattern P . Therefore, the result of the match

$\langle \text{"All"}, \text{"Men"}, \text{"Are"}, \text{"Mortal"} \rangle . \langle \text{"All"}, ?, \text{"Are"}, ? \rangle$ is the tuple $\langle \text{"Men"}, \text{"Mortal"} \rangle$.

When T_1 and P do not recursively match, that is, for some subexpression of T_1 , it is not equal to the corresponding (non-wildcard) subexpression of P , the result of $T_1.P$ is defined to be ε .

Ordering ($\mathbf{T}^* \times \mathbf{T}^* \rightarrow \{\text{TRUE}, \text{FALSE}\}$) A function that is absolutely essential to rules for secure logs—and which is therefore included in the core logic—is the “precedes” relation. $T_1 \prec T_2$ is defined as true if $T_1 \in L$ and $T_2 \in L$ and T_1 provably precedes T_2 in time. (Though this

relationship is represented in secure logs by a hash chain path between T_2 and T_1 , discussion of concrete implementations is avoided here. It is sufficient to make this operation available to the logic. The problem of finding this path is discussed in Section 5.3.2.)

Other functions ($(\mathbf{T}^*)^n \rightarrow \mathbf{T}^*$; $n > 0$) Arbitrary relations over tuple space may be necessary in order to express interesting rules. (For example, an application may wish to write rules that involve cryptographic operations, such as hash computation or signature verification.) Such functions are explicitly allowed in the language.

5.2.3 Logical expressivity

Sentences involving these relations are expressed in a bivalent predicate logic whose universal domain is \mathbf{T}^* .

Truth-functional connection For any formulæ Φ and Ψ , the truth functions are defined: negation ($\neg\Phi$), conjunction ($\Phi \wedge \Psi$), disjunction ($\Phi \vee \Psi$), and material conditional ($\Phi \rightarrow \Psi$).

Quantification The universal and existential quantifiers are also defined over \mathbf{T}^* , and have the form $(\forall\alpha) \Phi$ and $(\exists\alpha) \Phi$ respectively, where only the variable α is allowed to be free in Φ .

(The shorthand $(\forall\alpha \in S) \Phi$ represents the conditional expression $(\forall\alpha) ((\alpha \in S) \rightarrow \Phi)$. The set $S \subset \mathbf{T}^*$ can be defined using set-builder notation of the form $S = \{\alpha : \alpha \in L \text{ and } \Psi\}$, where the variable α may be free in the predicate Ψ .)

5.3 Algorithms used in Querifier

5.3.1 Evaluation of tuple logic expressions

To guarantee this tuple logic's decidability, quantification over infinite sets is not allowed. Intuitively, this restriction does not limit an application in the practical sense; the set of constraints an arbitrary application should wish to verify will only govern L (the logs under scrutiny), which is, by definition, a finite set. I now consider the problem of *evaluation*, that is, computing the results of expressions in this logic.

Logical formulæ

Each of the truth-functional connectives can be evaluated by a constant-time lookup in a small fixed truth table. A single quantification $(\forall \alpha \in S) \Phi$ or $(\exists \alpha \in S) \Phi$ can be evaluated by doing an exhaustive search of S for a witness which makes Φ false (in the former case) or true (in the latter case). At each iteration in the search, Φ must be evaluated once. Assuming Φ has only connectives and relations, then the un-optimized evaluation of a single quantifier is $O(n \cdot c)$, where n is the cardinality of S and c is the cost of evaluating the formula Φ (which may involve relation evaluation, described below). More generally, the complexity of any expression involving quantification is $O(n^d \cdot c)$, where d is the maximum depth of quantifier nesting.

Relations

Evaluating the pattern-match relation can be done using a recursive pairwise comparison algorithm mirroring the description given in Section 5.2.1. Any atom (character or ε) matches itself; the wildcard matches any tuple or atom. Two tuples T_1 and T_2 match if their lengths are the same and each respective pair of elements recursively match. This algorithm's complexity is $O(s)$, where s is the size of the pattern (more specifically, the number of sub-tuples and atoms in the pattern). Equality can be treated as a special case of match in which no wildcards are present.

Finding counterexamples and witnesses

Beyond merely determining whether a log is valid given a rule set, it is also important to consider how to identify the specific entries that are responsible for violations. This task is challenging because it is unclear, given an arbitrary expression, whether any individual subexpression's truth value is exceptional and therefore of interest. If a negation governs a quantifier, its meaning is inverted, making it impossible to automatically determine if, when a witness is found in the exhaustive search, it is evidence of *good* or *bad* behavior.

If automatic isolation of these witnesses is desired, rules may be transformed (preserving truth) such that negations only govern predicates, not quantifiers. This translation is performed by repeated application of De Morgan's law for truth-functions $\neg(\Phi \wedge \Psi) \Leftrightarrow (\neg\Phi \vee \neg\Psi)$; $\neg(\Phi \vee \Psi) \Leftrightarrow (\neg\Phi \wedge$

$\neg\Psi$)] and Quantifier Negation [$\neg(\forall\alpha)\Phi \Leftrightarrow (\exists\alpha)\neg\Phi$; $\neg(\exists\alpha)\Phi \Leftrightarrow (\forall\alpha)\neg\Phi$]. If, in the course of evaluating any universally quantified formula, a member of the domain set is found to make the quantified formula false, then this member serves as a *counter-example*. Likewise, a member which makes an existentially quantified formula true serves as a *witness*. Because quantifiers can be nested, such a witness or counter-example of a nested quantifier must be reported along with any bindings in effect from parent quantifiers for it to be meaningful.

5.3.2 Algorithms for ordering log entries

The graph of time

The hash chains in secure logs provide irrefutable evidence of order, so I turn now to the problem of determining the temporal relationship between any pair of entries in the log.

Determining order can naturally be cast as a graph search problem, because hash-chained log entries form a *graph of time*: a directed acyclic graph whose vertices are entries, and whose directed edges represent direct precedence. If a *hash chain path* exists in a log leading from entry B to entry A , then the event described by entry A must have happened before event B . (A corollary of the “happened before” relationship is potential causality: A may have affected B [67].) If instead a path exists from A to B , then B precedes A in time.

If neither of these directed paths exists, yet A and B are still a member of the same graph of time, they are contemporaneous events. They may not actually have happened simultaneously in “real” time, but the graph of time cannot establish their relative ordering; there exists only a common precedent, a common successor, or both.

In a system where concurrent events are impossible, the DAG degenerates to a list and such queries can be made very fast, but any interesting system will allow two events A and B to be contemporaneous and so a more general graph search must be performed.

The algorithms presented here represent various points in the time/space efficiency spectrum; space efficiency can be traded for time efficiency in varying degrees when solving this problem. Table 5.1 summarizes the complexity of the four techniques.

This analysis makes frequent reference to the following variables: the number of hosts k , the

Technique	Prep	Lookup	Space
Graph search	—	$O(e)$	$O(n)$
Full precomputation (Warshall's)	$O(n^3)$	$O(1)$	$O(n^2)$
Memoized graph search	—	$O(e)$	$O(n^2)$
Graph search with pruning	—	$O(e)$	$O(n)$
Precompute with timelines	$O(k \cdot e)$	$O(1)$	$O(n \cdot k)$

Table 5.1: Summary of graph-of-time search algorithms. “Prep” is the running time for the startup phase of the algorithm, and “Lookup” is the running time for each test of $A \prec B$. “Space” is the storage cost of the algorithm over all operations. (Complexity is given in terms of n log entries among k hosts and e edges in the graph of time.)

number of log messages in the system n , and the number of edges e . In all cases, the partial ordering of log messages is organized as a DAG representing the graph of time.

Graph search algorithms

Full graph search To determine if A precedes B , the algorithm performs a conventional graph search of the DAG starting at B and ending at A . $O(n)$ storage is required for the intermediate state (e.g., the frontier set) and the worst case time complexity is $O(e)$. While either depth-first or breadth-first search will give correct results, if the graph of time is structured as a set of long timelines with few intersections, depth-first search may consume a great deal of time searching the wrong branch for a desired entry.

Full precomputation Warshall’s algorithm for all-points shortest paths [102] can be used to calculate the pairwise ordering relationship for all pairs of messages in the DAG. This takes $O(n^3)$ time to compute, and storing this table takes $O(n^2)$ space, but subsequent tests for order reduce to table lookup and take constant time. Note that the table becomes invalid (and must therefore be recomputed) when new log entries are introduced; therefore, this technique is only recommended for static logs.

Memoized search A variant on full graph search, this technique optimizes for situations in which a few entries of interest are compared with many others. When evaluating $A \prec B$, the algorithm traverses the graph from B in search of A . Noting that every node x ever entering

the frontier set precedes B , the algorithm stores the relationship $x \prec B$ for future queries. Depending on the structure of the logs and query order, lookups may now return in constant time, though they still have a worst case running time of $O(e)$ as in conventional graph search.

I now consider a particular subset of all graphs of time: namely, those that comprise a small number of *timelines*. This occurs in distributed systems in which each participant's secure log establishes a total order on its entries. Such a per-host ordering may be efficiently verified by ensuring that each new message from a given host includes the hash of the previous message from the same host.

If this property holds for all hosts in the system, there exists an opportunity for some novel optimizations when searching the graph of time. The following two algorithms assume a system with k hosts (and therefore k timelines) such that that k is substantially less than the total number of log entries n .

Graph search with pruning First, assume that for each message x , its host $h = \mathbf{host}(x)$ is known, as well as its integer index in that host's log, $\mathbf{idx}(h, x)$. As such, to compute $A \prec B$, the algorithm begins as usual by searching the DAG from B with the intent of finding A . Note once again that for each x discovered during this search, $x \prec B$. If any message x is discovered such that $\mathbf{host}(A) = \mathbf{host}(x) = h$, we can then take advantage of the total ordering of the messages from h and compare $\mathbf{idx}(h, A)$ and $\mathbf{idx}(h, x)$:

- If $\mathbf{idx}(h, A) < \mathbf{idx}(h, x)$, then A precedes x in h 's log. Because $x \prec B$, it is the case that $A \prec B$ and so the search ends.
- If $\mathbf{idx}(h, A) = \mathbf{idx}(h, x)$, then $x = A$ because messages on h are totally ordered. Therefore, the search terminates, having shown $A \prec B$.
- If $\mathbf{idx}(h, A) > \mathbf{idx}(h, x)$, x is older than A . It is not yet known definitively whether $A \prec B$; a path might still exist from B to A . What has been shown is that, because x precedes A and time is acyclic, x cannot be on any such path. The algorithm may therefore *prune* this part of the search tree and continue searching from other nodes in the frontier set.

In the worst case, no vertices are pruned, resulting in the same space and time complexity as

graph search, but in graphs of time that result from very dense entanglement and few extended divergences, most search paths are pruned very quickly.

Precompute with timelines While a full precomputation of the pairwise ordering relationship consumes $O(n^2)$ space, some of this information is redundant when assuming that the DAG is composed of timelines. If $A \prec B$ for some entries A and B , it is also true that A precedes all subsequent entries in B 's timeline.

This means that for every entry x , on every host h there exists a unique index $i \in [0, \infty]$ such that, for any y in h 's timeline, if $i \leq \mathbf{idx}(h, y)$ then $x \prec y$. (An index of ∞ indicates no such entry exists.) This index is the least upper bound of the *projection* of x onto the timeline of host h ; that is, it is the “latest” that it may have happened from the perspective of h .

It is possible to precompute the full table \mathbf{P} of these projections: $\mathbf{P}(x, h) \leftarrow i$. Using this table, $x \prec y$ is true when $\mathbf{P}(x, h) \leq \mathbf{idx}(h, y)$, where $h = \mathbf{host}(y)$. Lookups in this table cost $O(1)$ and storage is $O(k \cdot n)$.

I now offer two approaches to pre-computing the table \mathbf{P} such that it maintains the invariant that $\mathbf{P}(x, h) \leq \mathbf{P}(y, h)$ for all $x \prec y$.

Naïve computation of P. The first approach exploits the fact that $\mathbf{P}(x, h) \leq \mathbf{idx}(h, y)$ iff $x \prec y$ and $h = \mathbf{host}(y)$. It begins by initializing $\mathbf{P}(y, h) \leftarrow \infty$ for every y on every host h ; this will be an initial estimate which will be relaxed during the precomputation. For every entry y at index $i = \mathbf{idx}(h, y)$ on its own timeline, the algorithm performs a DFS through the entire subgraph of preceding events x reachable from y . Each $x \prec y$, so the estimate relaxes: $\mathbf{P}(x, h) \leftarrow \min(\mathbf{P}(x, h), i)$. Once this search has been performed for every y on host h , $\mathbf{P}(x, h)$ will be equal to the smallest $\mathbf{idx}(h, y)$. The cost of this precomputation is $O(n + n \cdot e) = O(n \cdot e)$.

Improved computation of P. This technique operates similarly to the naïve approach, but reduces the cost of precomputation to $O(k \cdot e)$ by exploiting two insights:

- By iterating over all messages for each host in timeline order—that is, in order of *increasing* $\mathbf{idx}(h, y)$ —the estimate is relaxed no more than once: $\mathbf{P}(x, h)$ will be set to its lowest possible value immediately.

- Nodes may be pruned from the traversal when they cannot cause any further updates to \mathbf{P} . This happens whenever $\mathbf{P}(x, h) \leq \mathbf{idx}(h, y)$ —that is, when the search encounters a region of the graph that has already been identified as projecting to a point on the timeline of h that already precedes y .

Using this algorithm, each $\mathbf{P}(x, h)$ entry is only updated once. For each host h , then, each edge in the graph of time must therefore be followed exactly once; any attempt to follow an edge $x \rightarrow x'$ a second time would result in relaxing $\mathbf{P}(x, h)$ a second time. Therefore, the total number of edges visited per host is $O(e)$, and the total precomputation cost is $O(n + k \cdot e) = O(k \cdot e)$.

5.3.3 Incremental verification

An application may wish to use a rule evaluator not just as a *post facto* auditing tool but as a runtime watchdog. Such usage necessitates applying the verification engine “online,” on a growing log created by a running application. As an example, a supervisor console for a network of electronic voting machines could use incremental rule verification on its event log while the election is ongoing. A runtime rule violation might be cause to alert a poll worker to take the offending machine out of service for examination.

Unfortunately, so far I have only described algorithms for a verifier that operates “offline”: all at once, on a log considered to be complete. A straightforward approach to online verification is to periodically re-start the verifier from scratch, using as input the entirety of the log so far. This approach can prove quite costly; each time the verification process is begun, it will re-consider earlier entries that are unchanged from the last run. Clearly, a great deal of redundant computation may result, which, given complex rules involving multiple nested quantifications, will be polynomial in the number of log entries. If this performance penalty is avoided by evaluating rules more rarely, however, the ability to detect constraints violations as they occur is reduced.

Limitations of all-at-once evaluation

There is a more subtle limitation here: A post facto verifier makes no accommodation for potential future changes to the log. In particular, it cannot distinguish the difference between the case where an entry is missing from a partial log because it hasn't yet appeared and the case where it never will. Consider, again in the electronic voting context, the following constraint:

$$(\forall b \in L) (b.CAST_BALLOT \neq \varepsilon) \Rightarrow (\exists z \in L) (z.POLLS_CLOSED \neq \varepsilon \wedge b \prec\prec z)$$

This rule confirms that all ballots were cast before the polls closed on election day (and were not added later). The basic verifier, presented with an incomplete log (that is, one in which the polls-closed message has not yet appeared), will erroneously deduce a violation.

It is tempting to sidestep this particular complication by taking the additional manual step of segregating the rule set into two categories: those that may be evaluated at any time, and those (such as the foregoing) that must wait until the log is complete. But this is unsatisfying; certainly, a unified approach to rules and queries is preferable. Moreover, the verifier has enough information to know when a part of the evaluation is final and when it is not; it should be able to produce definitive answers as soon as they are available, rather than blindly waiting until the log is complete for these tricky cases.

Performing partial evaluation

Recall that $L \subset \mathbf{T}^*$ is the set of all log messages under scrutiny. Note that L grows monotonically—log entries, once added, are never taken away from the log. (If this were possible, any secure logging scheme would be faulty.) For this reason, $S \subseteq L$ is defined to be *open* if new elements may appear later, and *closed* if S contains all the elements that it will ever contain. The evaluation of a quantification over a closed S , then, behaves precisely as before. The evaluation of a quantification over an open S , however, behaves differently. In the case where, while evaluating $(\exists \alpha \in S)\Phi$, a witness is not found, it cannot be assumed that one will never be found. Likewise, in the case where, while evaluating $(\forall \alpha \in S)\Phi$, a counter-example is not found, it cannot be assumed that one will never be found.

To represent this, any particular evaluation which involves quantification over an open set may result in a *reduction* rather than in a result. This reduction, while its truth value is unknown, is a simplification of the original problem. That is, when the reduction is evaluated, no computation will be repeated in the search for truth.

In the case where no witness is found for an existential quantifier over an open set, the reduction returned will only represent the computation *yet to be done* (i.e., the evaluation of the governed formula in the case where the variable is bound to any *future* set member.) In the case where no counter-example is found for a universal quantifier over an open set, the reduction returned will similarly only represent computation regarding future entries. In all other cases, evaluation of quantification over an open set behaves exactly as it does over a closed set (e.g., if a witness is found in an existential over an open set, there is no reason not to evaluate this as true, even though the set is open). Now, rather than $O(n \cdot c)$ for each quantifier evaluation (given cost c for the quantified expression), this becomes $O(c)$ per incremental evaluation in the worst case.

Because these reductions have been introduced as placeholders for truth values, truth-functions and quantifiers must be adjusted to accommodate sub-formulæ that evaluate to reductions (i.e., the cases where truth-functions govern quantifiers or when quantifiers are nested). For the truth-functions, the truth tables used for the evaluation of each truth-function are converted to the corresponding truth table for a three-valued system (where the third value is the unknown value). In the case where the evaluation of some truth-function which connects two formulæ cannot be known because reductions are evaluated from either or both of the connected formulæ, these reductions are connected using this truth-function and returned. Similarly, the reductions which are returned by the evaluation of nested quantifiers will remember which of its evaluations returned reductions, and therefore need to be re-evaluated. Concisely stated, incremental evaluation only saves work on the inner-most nested quantifier which governs the open set in question. In the case where d quantifiers are nested over the same open set, the run time for each incremental evaluation is $O(n^{d-1} \cdot c)$ per incremental evaluation in the worst case, with $O(n^{d-1})$ space needed to save the reduction.

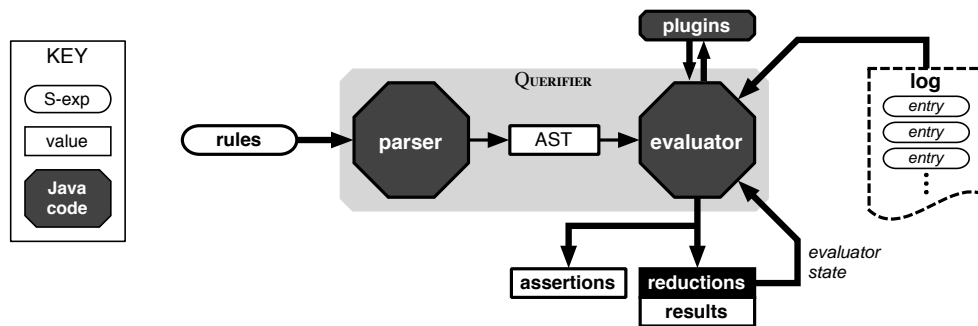


Figure 5.1: Querifier components and operation. Applications supply rules in the form of S-expressions; the verifier parses rule expressions into an AST suitable for evaluation. Log entries, also S-exps, are fed to the verifier, which recursively interprets the AST for each, finally yielding a result value and a list of assertions (if any). Partial results contain sufficient state to resume the evaluator without performing redundant computation when new log data arrives.

5.4 Implementation

5.4.1 Introduction

Owing to the need, discussed in Section 5.3.3, for an online verifier that can be embedded into VOTEBBOX, solutions involving dedicated relational database engines or theorem provers were considered to be impractical. Instead, Kyle Derr and I developed Querifier, a log verification tool comprising approximately 2700 semicolons of Java source (including tests and performance measurement code). Its key advances are:

Expressivity. Any rule expressible in the predicate logic of Section 5.2 is also directly expressible in the rule language understood by Querifier; converting from one to the other is a straightforward syntactic transformation (from infix logical connectives to prefix S-expression notation).

Incremental evaluation. Querifier implements the incremental evaluation algorithm described in Section 5.3.3, and is therefore able to offer partial results with low overhead per query.

5.4.2 Operation

The structure of Querifier is summarized in Figure 5.1. The core Querifier implementation comprises a rule parser and evaluator. Rules and log data are represented in a format based on Rivest’s canonical S-expression encoding [77]. This format was chosen due to its compact representation, low scanning

$ \begin{aligned} &(\exists x \in L) \\ &(\exists y \in L) \\ & \left(\begin{aligned} &(x.\text{POLLS_OPEN_MSG} \neq \varepsilon) \\ &\wedge (y.\text{POLLS_CLOSED_MSG} \neq \varepsilon) \\ &\wedge (x \prec\prec y) \end{aligned} \right) \end{aligned} $	<pre> (exists x all-set (exists y all-set (and (match POLLS_OPEN_MSG x) (match POLLS_CLOSED_MSG y) (precedes x y all-dag)))) </pre>
---	---

Figure 5.2: S-expression representation of logical rules. The simple rule here asserts that there exist both a polls-open and polls-closed message in the log, and that the former precedes the latter. The special value `all-set` is the set of the available log messages (corresponding to the finite set L in the logic), and `all-dag` is a DAG of time constructed from `all-set` by an application plugin.

complexity, and ability to encode arbitrary recursive data structures (particularly tuples, as defined in Section 5.2). The canonical encoding of an S-expression is unambiguous and therefore suitable for digital signatures and hashing. It is straightforward to marshal structured log data into and out of S-expressions, and applications may even choose to directly use S-expressions when representing secure logs.

Clients of Querifier initially supply *rules* in the form of a single S-expression representing a sentence in predicate logic. The sentence can be arbitrarily complex; typically, sets of rules are conjoined to form a single rule expression. The complete grammar is omitted here for space, but an example transformation is given in Figure 5.2. Rules are parsed, according to the grammar, into an abstract syntax tree (AST).

Applications can also supply *plugins*, small pieces of Java code that define additional functions for use in rules (see Section 5.2.1). A common function of all plugins is to identify which portions of each log message correspond to hash chain pointers so that the DAG of time may be constructed and the “precedes” operation can be computed during rule evaluation.

Thus initialized, Querifier is ready to consume log data (as S-expressions) and generate results. When invoked, the rule *interpreter* recursively navigates the AST, using the algorithms described in Section 5.3 to compute a result value for the rule.

While log data is still being introduced by the application, the log is considered “open” and some quantifiers may return, instead of truth values, *reductions* as defined in Section 5.3.3. These objects contain sufficient state to resume computation at any time without duplicating effort. The log is

eventually “closed” by the application, signaling to Querifier that no more log data will appear; from this point forward, the final verification result is computable and no reductions will ever be returned.

In the next section I present an early performance evaluation of Querifier on realistic data and rules. Experiments cover variations on the evaluator implementation, including all-at-once versus incremental evaluation, as well as several graph search algorithms for determining order.

5.4.3 Experimental setup

The performance of Querifier was measured on a synthetic VOTEBOX election log and a set of rules that are representative of a realistic deployment of the Auditorium polling place.

Voting simulation. The log, comprising 763 individual events from 9 nodes (eight voting booths and one supervisor console), was collected during an 8-hour real-time simulation of an election held in a single polling place. The simulation was generated using a modified version of VOTEBOX, replacing the supervisor and voter GUIs with automated drivers that behave as follows. After opening the polls, the supervisor authorized a new ballot (simulating a new voter being assigned to a voting machine) every 10 to 120 seconds when voting machines were available. Each “booth” node simulated a voter who completed his/her ballot anywhere from 30 to 300 seconds later. After eight hours, the polls were closed; a total of 127 ballots were cast in that time.

Voting rules. The experimental rule set contains seven constraints, expressed in English as follows:

1. All messages are correctly-formatted Auditorium voting messages.
2. There exists a `polls-open` message beginning the election.
3. There exists a `polls-closed` message concluding the election.
4. The `polls-open` precedes the `polls-closed`.
5. Every `cast-ballot` is preceded by an `authorized-to-cast`, and their authorization nonces match.
6. Every `cast-ballot` precedes a `ballot-received`, and their authorization nonces match.
7. Every `cast-ballot` has a unique authorization nonce.

This set of rules is directly derived from the goals set forth in Chapter 3, namely to clearly identify the set of valid votes (occurring on election day) so that they may be counted correctly. The Audi-

torium voting protocol was designed with these goals in mind, and so incorporates the sequence of messages described above. Therefore, the rules check whether the Auditorium voting protocol is being followed correctly and identify appropriate votes to be tallied. Other correctness properties may be of interest; for example, in an auditing scenario, a particular query (“which votes were cast on machine X?” “which machines joined the network late in the day?”) might be written *ad hoc* and fed to Querifier in order to answer a specific question about an election transcript.¹

Rule 1 above, expressed in predicate logic, is quite simple:

$$(\forall x \in L) (a.AUDITORIUM_MESSAGE \neq \varepsilon)$$

Rules 5, 6, and 7 may be represented as three separate logical expressions or combined into the expression:

$$\begin{aligned} (\forall b \in L) (b.CAST_BALLOT \neq \varepsilon) \Rightarrow (& \\ (\exists a \in L) (a.VOTE_AUTH \neq \varepsilon \quad \wedge \quad a.AUTH_NONCE = b.BALLOT_NONCE \quad \wedge \quad a \prec\prec b) & \\ \wedge \quad (\exists r \in L) (r.VOTE_RECEIPT \neq \varepsilon \quad \wedge \quad r.RECEIPT_NONCE = b.BALLOT_NONCE \quad \wedge \quad b \prec\prec r) & \\ \wedge \quad \neg(\exists x \in L) (x.CAST_BALLOT \neq \varepsilon \quad \wedge \quad x.BALLOT_NONCE = b.BALLOT_NONCE \quad \wedge \quad x \neq b)) & \end{aligned}$$

A straightforward translation from the above to S-expression syntax (as described in Section 5.4) yields rules that Querifier can directly evaluate. Note that the maximum nesting depth of quantification is 2; by the reasoning in Section 5.3, a naïve implementation will perform $O(n^2)$ computations.

Equipment. The machine running Querifier was a lightly-loaded dual 2 GHz PowerPC G5 workstation (Mac OS X 10.4) with 4 GB RAM; the Java runtime in use was the Sun Java HotSpot Client VM (1.5.0).

¹The broader question of what it means to have a “correct election” is substantially more vague and is beyond the scope of this chapter, and indeed this thesis, which carves out a few specific notions of correctness and offers ways to test that correctness (Auditorium logs, Querifier queries, and cryptographic challenges).

5.4.4 Results

Incremental evaluation. When given the entire 763-entry log at once, the basic Querifier implementation completed rule verification in about 220 CPU seconds (0.3s per log entry). VOTEBOX, however, requires a solution that can be used many times during an election, because administrators would ideally like to know about obvious violations within a short amount of time (that is, before the end of the day).

A naïve approach is to re-run Querifier anew after every message. The cost, of course, is unacceptable: the last run takes the same 220 seconds, resulting in an overall computation time of just over 30 thousand seconds, or about 8½ hours—longer than the election itself!

Much of this computation is redundant, and as described in Section 5.3.3, an incremental approach is vastly preferable. To demonstrate this, I simulated an online verification scenario with a persistent instance of incremental Querifier using reduction-based incremental evaluation. Entries from the synthetic log were fed to this instance with different batch sizes ranging from one (invoking the verifier for every entry) through the entire log (essentially the offline verification case). For comparison, I also attempted to use a non-incremental version of Querifier with each partial log at the same event intervals (simulating the effect of using an offline, all-at-once verifier in an online context). Figure 5.3 shows the dramatic difference between the performance of full vs. incremental evaluation at various intervals.

Graph search. Another experiment compares three of the algorithms for computing order between entries in the graph of time: BFS, memoized BFS, and BFS with timeline pruning (see Section 5.3.2). The performance of these algorithms was examined in the incremental verifier, using the same batch-size variation described previously; the results are shown in Figure 5.4.

The memoized search algorithm can be seen to perform substantially better when fed large amounts of new log data at once; this is because the first precedence test in the new dataset frequently ends up pre-computing other ordering relationships that will soon be requested. Pruned BFS performs better still, and does so consistently for any size of input data; this can be attributed to the few (9) distinct timelines that make up the overall graph of time, and systems with a greater number of participants

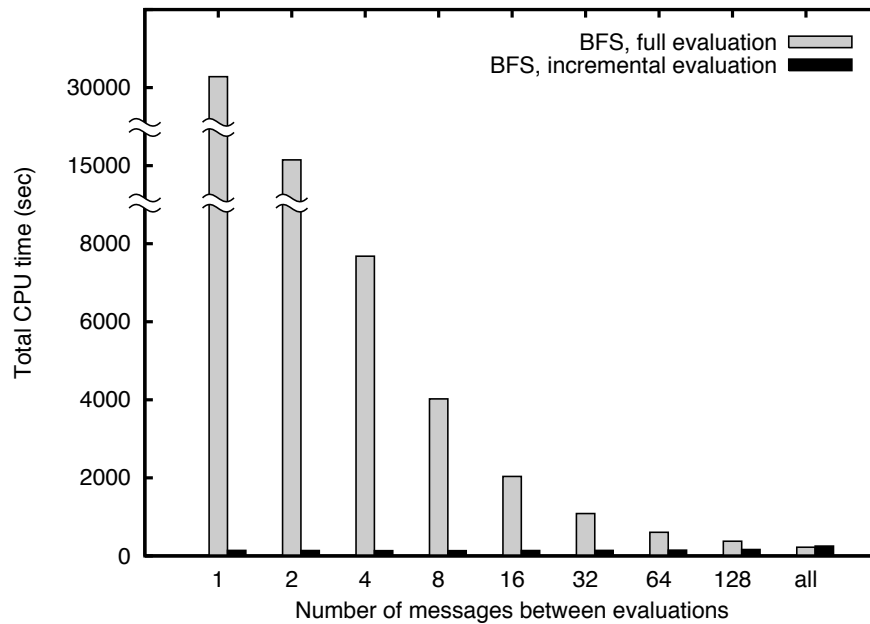


Figure 5.3: Incremental evaluation. Bars indicate total time to consume and evaluate the entire log. The rightmost bar represents an interval equal to the length of the input, in which case the two approaches are equivalent; as the intervals get shorter, the costs of re-verifying from scratch become obvious.

will generate logs that cannot be pruned as quickly. (Note that if there is no total ordering of each participant’s messages, this algorithm is not applicable.)

Summary of implementations. Figure 5.5 uses a log scale to collapse the many orders of magnitude separating these implementation variants. Optimized graph search algorithms improve upon all-at-once verification, but cannot fundamentally change the problem’s complexity as the reduction-based incremental approach does. Finally, note that at an increment of 763 events (the full log), incremental and full verification have almost identical performance (there is a small amount of overhead involved in the additional data structures necessary for incremental operation, though they are hardly used). These evaluations show that optimization in the verifier can yield excellent gains, to the point that the entire voting log can be processed in about 30 total CPU seconds (0.04s per entry).

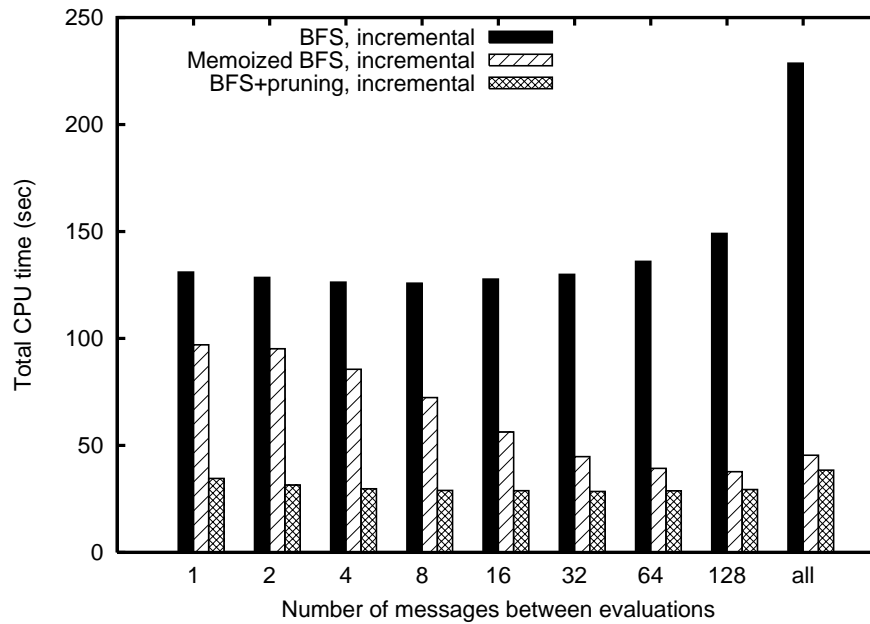


Figure 5.4: Graph search. Incremental verification performance across several graph search algorithms (described in Section 5.3.2). As in Figure 5.3, the verifier is re-run at various intervals to quantify the overhead associated with each invocation. Bars indicate total time to verify the entire log.

Ruleset comparison. Returning to the voting scenario, the only nodes that need perform “online” verification are the supervisors, and they can be provisioned appropriately. If performance is an issue, the verification interval can be increased and results obtained less frequently. Likewise, not every rule must necessarily be evaluated every time a new message arrives. Figure 5.6 compares the performance of two smaller rule sets to the full rule set. Finally, the algorithmic complexity of the “online” ruleset can be reduced (for example, to singly-nested quantification) so that some simpler rules (for example, “has the election begun?”), allowing simple verifications to be executed even on underpowered devices or devices with much higher rates of log activity.

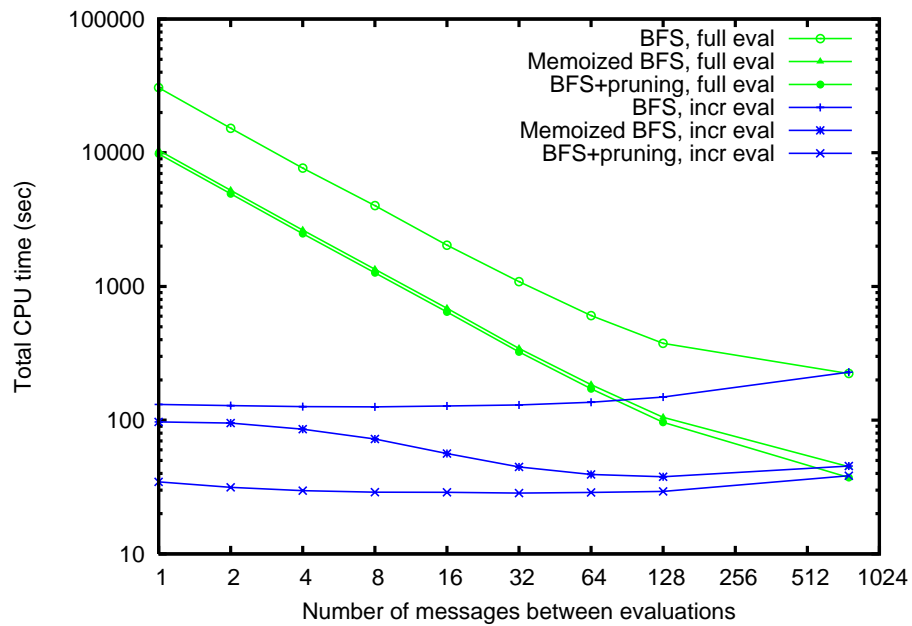


Figure 5.5: Summary of implementations. As before, when the batch size equals the entire log, full and incremental evaluation are equivalent, but when results are requested after every entry, incremental evaluation is two orders of magnitude less expensive than full evaluation.

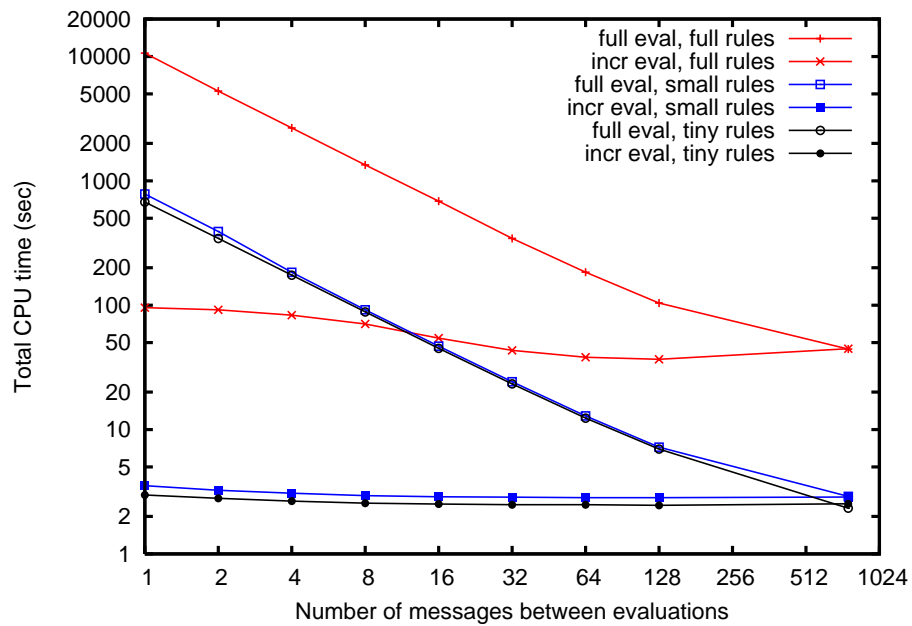


Figure 5.6: Ruleset comparison. The “full” set of rules is the same as in earlier figures. The “small” set includes only rules 1–4; the “tiny” set comprises 1 and 2. (All evaluators used the memoized graph search algorithm for precedence queries.)

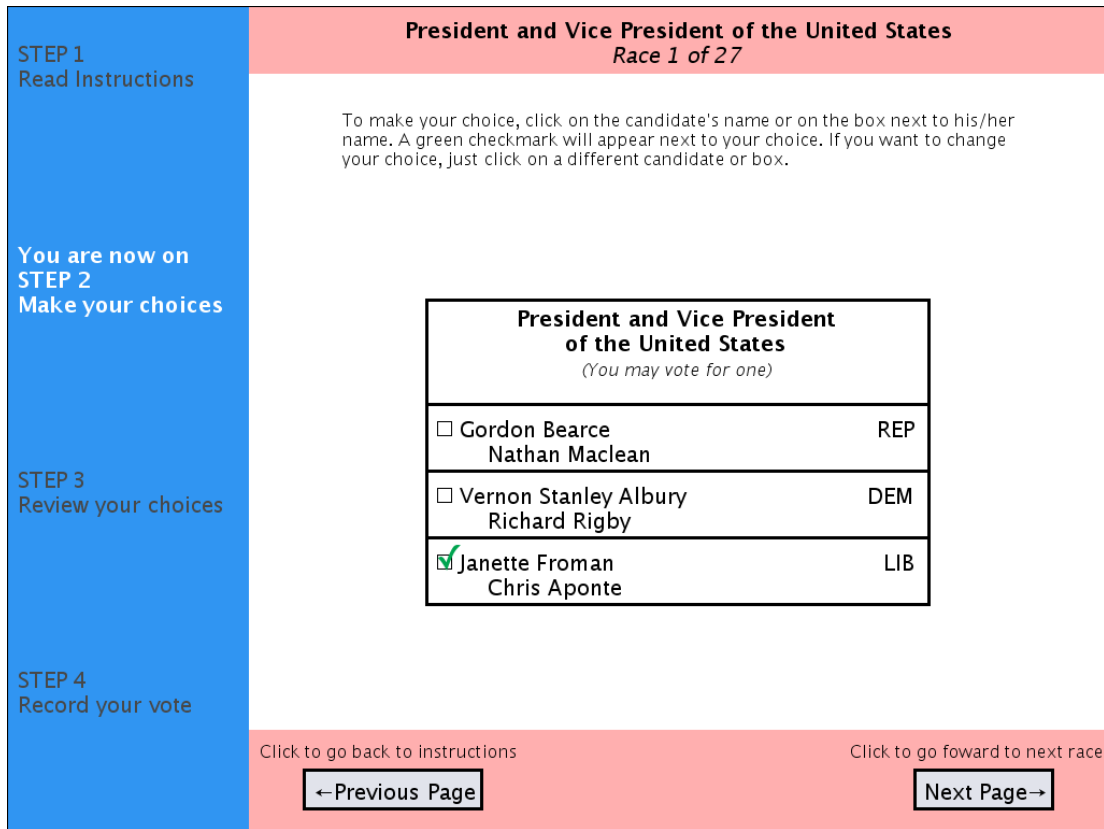
CHAPTER 6

PROPERTIES OF THE USER INTERFACE

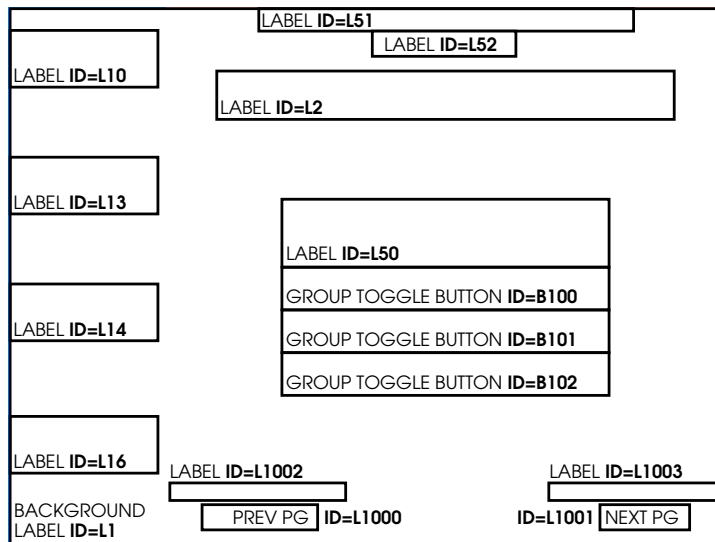
6.1 Pre-rendering for assurance

A recent study [33] bolsters much anecdotal evidence suggesting that voters strongly prefer the DRE-style electronic voting experience to more traditional methods. Cleaving to the DRE model (itself based on the archetypical computerized kiosk exemplified by bank machines, airline check-in kiosks, and the like), VOTEBOX presents the voter with a ballot consisting of a sequence of *pages*: full screens containing text and graphics. The only interactive elements of the interface are *buttons*: rectangular regions of the screen attached to either navigational behavior (e.g., “go to next page”) or selection behavior (“choose candidate *X*”). (VOTEBOX supports button activation via touch screen and computer mouse, as well as keyboards and assistive technologies). An example VoteBox ballot screen is shown in Figure 6.1;

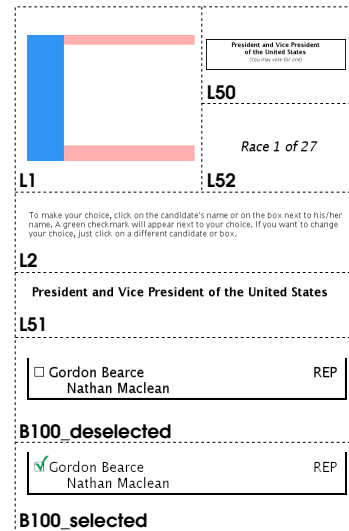
This simple interaction model lends itself naturally to the pre-rendered user interface, an idea popularized in the e-voting context by Yee’s *Pvote* system [104, 105]. A pre-rendered ballot encapsulates both the logical content of a ballot (candidates, contests, and so forth) and the entire visual appearance down to the pixel (including all text and graphics). Generating the ballot ahead of time allows the voting machine software to perform radically fewer functions, as it is no longer required to include any code to support text rendering (including character sets, Unicode glyphs, anti-aliasing), user interface element layout (alignment, grids, sizing of elements), or any graphics rendering beyond bitmap placement.



(i)



(ii)



(iii)

Figure 6.1: Sample VOTEBX page. The voter sees (i); a schematic for the page is shown in (ii); a subset of the pixmaps used to produce (i) are shown, along with their corresponding IDs, in (iii).

More importantly, the entire voting machine has no need for any of these functions. The only UI-related services required by VOTEBOX are user input capture (in the form of (x, y) pairs for taps/clicks, or keycodes for other input devices) and the ability to draw a pixmap at a given position in the framebuffer. We therefore eliminate the need for a general-purpose GUI window system, dramatically reducing the amount of code on the voting machine.

In our pre-rendered design, the ballot consists of a set of image files, a configuration file which groups these image files into pages (and specifies the layout of each page), and a configuration file which describes the abstract content of the ballot (such as candidates, races, and propositions). This effectively reduces the voting machine's user interface runtime to a state machine which behaves as follows. Initially, the runtime displays a designated initial page (which should contain instructional information and navigational components). The voter interacts with this page by selecting one of a subset of elements on the page which have been designated in the configuration as being selectable. Such actions trigger responses in VOTEBOX, including transitions between pages and commitment of ballot choices, as specified by the ballot's configuration files. The generality of this approach accommodates accessibility options beyond touch-screens and visual feedback; inputs such as physical buttons and sip-and-puff devices can be used to generate selection and navigation events (including "advance to next choice") for VOTEBOX. Audio feedback could also be added to VOTEBOX state transitions, again following the Pvote example [104].

6.2 Ballot creation

In order to construct these complex pre-rendered ballots, VOTEBOX includes a separate ballot creation tool intended to be run by election administrators well in advance of an election. The Ballot Creator, a graphical Java program, is the final destination of the layout and rendering logic that is omitted from VOTEBOX. The ballot creation tool, therefore, is the only piece of software that needs to understand all the jurisdictional peculiarities; it acts as a sort of compiler, generating a ballot description that is "executed" at "run time" (election day) by the VOTEBOX system itself. This greatly simplifies the software certification process, as testing labs need consider just a single version of VOTEBOX rather than separate versions customized for each state's needs.

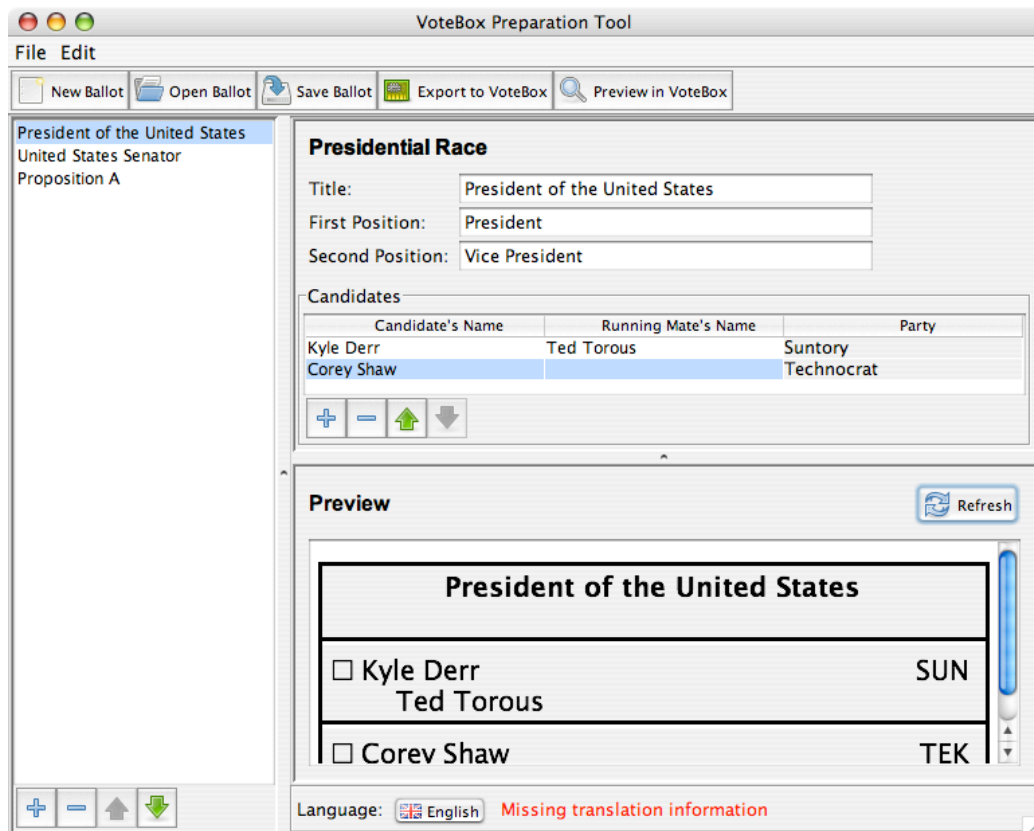


Figure 6.2: The VOTEBOX ballot creator. Visible in this screenshot is the main window, split into three panes: the list of contests (left), the contest editor (right top), and the contest preview (right bottom).

The user interface for the ballot creator, which is a conventional Java Swing graphical application, is shown in Figure 6.2. Contests (races or propositions) are created using the buttons at bottom left; information about candidates (for races) or propositions is entered in the editor pane, and the actual pre-rendered graphics are shown in the preview pane. The ballot can be *exported* to the format expected by VOTEBOX; this is the final pre-rendering step and will result in a Zip archive of XML instructions and PNG graphics. Because information is lost in this step (candidate names are represented as bitmaps, not as Unicode strings), the ballot can also be saved to a different (lossless) file format that represents the current ballot editing session in progress; the creator can reconstruct its working state from this file. The creator knows how to create ballots that include multiple languages

simultaneously; note that in Figure 6.2, a warning is visible because not all information has been localized (translated to each language expected in the ballot).

The ballot creator is a useful tool for creating VOTEBOX ballot representations, but these two implementations are entirely decoupled. Any other mechanism could be used to create valid ballot files; similarly, a compatible implementation of the VOTEBOX frontend could be developed—possibly with substantially differing internal details from those described in Chaps. 3 and 4—that would still use the same ballot format to present an identical experience to the end user.

6.3 Human factors experimentation

Also of note is the utility of this separation for our human factors collaboration. From the outset, VOTEBOX has been a collaborative effort not just within the Rice Computer Security Lab¹ but with faculty and students working in the Rice Computer-Human Interaction Laboratory (CHIL) Head: Mike D. Byrne; <http://chil.rice.edu>. Kyle Derr, Ted Torous and I worked with Kristen Greene, and Sarah Everett of CHIL to turn their experimental requirements into a user interface specification and, eventually, working code in VOTEBOX and the ballot creator. The result is shown in Figure 6.1.

In addition to the graphic and interaction design of the ballot, the human factors experimentation proposed by CHIL required additional data collection for each subject. Data of interest included time spent on each screen, overall completion time, plaintext of the subject's selections, and so on. (See Section 7.2.2 for a discussion of the security and privacy implications of these additional data.)

While the two research groups worked to develop and refine the user interface and data collection requirements, Derr, Torous and I were able to design and develop the necessary VOTEBOX infrastructure concurrently, having established early on a general enough PRUI framework to accommodate nearly any ballot design that might eventually be settled upon. In fact, VOTEBOX has a number of UI capabilities not used by the current ballot, including:

- **Alternative inputs**, such as hardware buttons and other (non-touchscreen, non-mouse) assistive devices. This support includes an associated hidden navigation order between UI elements,

¹Head: Dan S. Wallach; <http://seclab.cs.rice.edu>

necessary for systems with “next” and “previous” inputs, such as scroll wheels, sip-and-puff systems, and the like.

- **Focus indicators**, represented as an additional pair of states (*focused-but-not-selected*; *focused-and-selected*) for each UI element that can be navigated to, either by next/previous inputs (see above) or by “mouse over” events.

This loose UI coupling is an unexpected strength of the PRUI approach: the runtime model (display graphics, accept clicks and other input) is simple enough that it assumes very little about the specific user experience it will eventually deliver. Of course, this complexity must reside *somewhere*, and as was stated above, this place is the ballot creator. Indeed, our up-front work on the creator was substantial, requiring multiple rounds of iterative design with the CHIL researchers to get it right. Yet, while this early work was substantial, most subsequent changes in HCI experimental design required only modest changes to the ballot creator, and usually none at all to VOTEBOX itself. This permitted us to take more care with changes to VOTEBOX, which is the code artifact that had to be most stable for use in human factors experiments and security research. Using a PRUI approach helped erect a firewall of sorts between the rapidly-changing ballot design and the less mercurial voting machine itself. I expect that this benefit will apply equally to any production e-voting system.

CHAPTER 7

THE VOTEBOX PROTOTYPE

7.1 Introduction

The VOTEBOX project was launched by our lab¹ in February 2006, with software development beginning in May. In three years it has been the subject of several refereed publications [87, 86, 84, 88, 33] and figures prominently in the Ph.D. thesis of Sarah Everett [34] (as well as this document). It has been ported to two language runtimes and three graphics APIs, and has been used in several human factors experiments involving hundreds of participants.

This chapter describes the project and its resulting software artifacts in detail. In Section 7.2, I recount several relevant metamorphoses in the history of the project, and in Section 7.3 provide some measurements of the current implementation as of 2009.

7.2 Software implementation notes

7.2.1 Secure software design

From the beginning of the project, it was the group's intent to develop a research platform to explore both security and human factors aspects of the electronic voting problem. Our early directions included:

1. Reduced trusted code base through use of a pre-rendered user interface, inspired by Yee [105]
2. Software simulation of hardware-enforced separation of components, based on Sastry et al. [90]

¹The Rice Computer Security Lab, <http://seclab.cs.rice.edu/>, run by Dan Wallach.

3. Hardware support for strict runtime software configuration control (i.e., trusted computing hardware)
4. Recoverable and secure audit logs

The first approach, `PRUI`, became a core part of the software design of `VOTEBOX`, as documented in Chapter 6. From the Sastry example we borrow the technique of enforcing inter-voter privacy by tearing down and reconstructing large parts of the program between voters.² The `votebox.VoteBox` class, a singleton object responsible for bootstrapping the Auditorium and user interface subsystems, essentially “reboots” the latter between voters. The user interface is not instantiated until an `authorized-to-cast` message from the supervisor console is received by the local Auditorium node. At this point, the voting state machine (`votebox.driver.Driver`) is constructed, which in turn uses the correct view factory to construct the objects necessary to interact with the display technology in use (SDL or AWT; see below). The Driver also constructs an instance of `votebox.middle.ballot.Ballot` to extract from the ballot definition file the ballot state machine and layout information (see Chapter 6).³ The driver, view, and ballot objects are discarded (pointers destroyed, objects scheduled for garbage collection) when the ballot is cast or challenged. The Java runtime enforces that these freed objects can no longer be read by running code, preventing information from leaking from one voter to the next.

The topic of *trusted hardware* drove a substantial number of early design decisions about `VOTEBOX`'s target platform. The original strategy for software configuration assurance was to develop a voting system to run on the Xbox 360 video game platform,⁴ initially developing `VOTEBOX` in managed C# (that is, C# code using the Microsoft XNA Framework and targeting the managed runtime of the Xbox). We theorized that a video game console and an electronic voting machine shared a number of features:

²For pragmatic reasons—chiefly, a focus on writing straightforward, maintainable code for those auditing (and developing!) the code—`VOTEBOX` does not apply the other technique from Sastry et al. (namely, observable “wires” between distinct modules), but this is an interesting direction for future work on the `VOTEBOX` platform.

³The objects used to capture a voter's selections in `VOTEBOX` exemplify the classic Model-View-Controller design pattern for interactive applications: the ballot is the model, the View is (unsurprisingly) the view, and the Driver mediates between them. This MVC system is constructed anew for each voter by the `VoteBox` object, which receives the completed `Ballot` object to be cast and hands it to the Auditorium code. In a sense, this represents another complete MVC system: the `Ballot` remains the model, but now the network is the “view” that is tied to it by the `VoteBox` acting as a controller.

⁴The `VOTEBOX` name derives in part from this early direction, known at the time as the “`BALLOTBOX 360`”.

1. The Xbox (both the original and the 360 update) has sophisticated hardware devoted to ensuring that the system runs only certified software programs, which is an obviously useful feature for a DRE.⁵
2. Video game systems are designed to be inexpensive and to withstand some abuse, making them good candidates for use in polling places.
3. A lack of a conventional desktop operating system is no problem for a prerendered user interface; we were fairly confident that an Xbox could handle displaying static pixmaps.

Derr, Torous, and I consequently developed the first VoteBox prototype as a conventional .NET desktop application, intending to port it to XNA when the toolchain and hardware became available. Hardware was not forthcoming, however, and being able to test only on Windows systems (.NET support on other platforms via the open-source Mono project not being mature enough to support our code) proved a development and deployment bottleneck. We found that development for a more widely-available software platform was both easier for us and more likely to result in a usable research product.

By August, 2006 we had ported our early VOTEBOX prototype to Java. We had no intention of relying on Java's AWT graphical interface (and its dependency, in turn, on a window system such as X or Windows). Instead, we intended to develop VOTEBOX atop SDL, the Simple DirectMedia Layer,⁶ a dramatically simpler graphics stack. (The Pvote system also uses SDL as a side-effect of its dependency on the Pygame library [104].) Regrettably, the available Java bindings for SDL suffered from stability problems, forcing us to run our PRUI atop a limited subset of AWT (including only bitmap drawing and user input events).

Finally, the impetus for a robust auditing infrastructure resulted directly from my experience with the contested Webb County primary election in Laredo, in the context of my recently-completed work on peer-to-peer publishing systems [85]. The details of that field work, and the Auditorium system

⁵Of note: the hardware protection scheme on the Xbox was broken by Huang, whose book on the subject [49] would of course be available to any potential hacker. There was some concern among the group that (as there is quite a bit of interest in running unsanctioned/pirated game software on Xbox) it was only a matter of time until the 360 fell as well; indeed, by May 2006 there had already been rumors and video circulated of a firmware hack that would allow unauthorized software to run on the 360. [97]

⁶<http://www.sdl.org>

that resulted, are fully explained in Chapter 3.

7.2.2 *Insecure software design*

As mentioned above, we intended from the beginning that VOTEBOX would serve as a foundation for e-voting research of different stripes, including human factors studies. The specifics of this collaboration are described in Chapter 6.

The version of VOTEBOX used in these studies is modified to emit fine-grained data tracking the user's every move: the order of visited screens, the time taken to make choices, and so forth. This sort of functionality would be considered a heinous breach of voter privacy in a real voting system, so we took great pains to make very clear the portions of the code that were inserted for human factors studies. Essential portions of this code were sequestered in a separate module that could be left out of compilation to ensure that no data collection can happen on a "real" VOTEBOX; later we made this distinction even more stark by dividing the VOTEBOX codebase into two branches in our source control system.

Of course, a well-known hazard of maintaining multiple long-lived branches of a software artifact is the difficulty of keeping improvements (including critical bug fixes) appropriately synchronized between them. Therefore, in 2008, VOTEBOX was painstakingly merged into a single unified branch. To retain the ability to reliably compile "evil" (instrumented) and "good" (non-instrumented) versions at will, I (working in the Security Lab with Kevin Montrose) developed a source-to-source Java translator called JPP⁷ and used it to create conditional-compilation regions for human-factors code.

It is noteworthy that some of the most interesting human factors results [34, studies 2 and 3] require a voting machine that is *malicious* (beyond simply capturing privacy-violating timing data). One study measured how likely voters are to notice if contests are omitted from the review screen; another, if votes on the review screen are *flipped* from the voter's actual selection. We labeled this *evil code* with alarming language (including adding the word "evil" to the names of relevant classes and methods) as well as wrapping it in JPP conditional-compilation regions so that, as with the data collection code, there would be no confusion in either code auditing or compilation scenarios.

⁷By analogy with `cpp`, the ubiquitous preprocessor for the C programming language.

7.2.3 Concrete representation of data

When it came time to develop the Auditorium network protocol, we chose to use a subset of the S-expression syntax defined by Rivest [77]. Previous experiences with peer-to-peer systems that used the convenient Java `ObjectOutputStream` for data serialization resulted in protocols that were awkwardly bound to particular implementation details of the code, were difficult to debug by observation of data on the wire, and were inexorably bound to Java.

S-expressions, in particular the canonical representation used in Auditorium, are a general-purpose, portable data representation designed for maximum readability while at the same time being completely unambiguous. They are therefore convenient for debugging while still being suitable for data that must be hashed or signed. By contrast, XML requires a myriad of canonicalization algorithms when used with digital signatures; we were happy to leave this large suite of functionality out of VOTEBOX.

We quickly found S-exps to be convenient for other portions of VOTEBOX. They form the disk format for our secure logs (as carbon-copies of network traffic, this is unsurprising). Pattern matching and match capture, which we added to our S-exp library initially to facilitate parsing of Auditorium messages, subsequently found heavy use at the core of Querifier, our secure log constraints checker, allowing its rule syntax to be naturally expressed as S-exps. (Querifier is the subject of Chapter 5.) Even the human factors branch of VOTEBOX (Chapter 6) dumps user behavior data in S-expressions.

7.3 Metrics

7.3.1 Code size

Table 7.1 lists several code size metrics for the modules in VOTEBOX, including all unit tests. We aspired to the compactness of Pvote's 460 Python source lines [104], but the expanded functionality of our system, combined with the verbosity of Java (especially when written in clear, modern object-oriented style) resulted in a much larger code base. The `votebox` module (analogous to Pvote's functionality) contains nearly twenty times as many lines of code. The complete VOTEBOX codebase, however, compares quite favorably with current DRE systems, making thorough inspection of the

module	semicolons	stripped LOC
sexpression	1170	2331
auditorium	1618	3440
supervisor	959	1525
votebox	3629	7339
	7376	14635

Table 7.1: Size of the VOTEBX trusted codebase. *Semicolons* refers to the number of lines containing at least one ‘;’ character and is an approximation of the number of statements in the code. *Stripped LOC* refers to the number of non-whitespace, non-comment lines of code. The difference is a crude indicator of the additional syntactic overhead of Java. Note that the ballot preparation tool is not considered part of the TCB, since it generates ballots that should be audited directly; it is 4029 semicolons (6657 stripped lines) of Java code using AWT/Swing graphics.

source code a tractable proposition.

7.4 Performance

By building a prototype implementation of our design, we are able to validate that it operates within reasonable time and space bounds. Some aspects of VOTEBX require “real time” operation while others can safely take minutes or hours to complete.

The Auditorium design, despite its apparently high computational and bandwidth requirements, is entirely tractable for a network the size of a typical polling place. For the following performance estimates, I assume that an individual VOTEBX casts at most one ballot every 3 minutes. This is an extremely high rate of ballot casting; it is hard to imagine a single voting machine continuously sustaining a rate of 20 voters per hour. Therefore, this is an excellent upper bound for our estimates. A polling place will be assumed to have 10 VOTEBXes, which is the highest concentration of electronic machines per polling place in the United States found in a 2004 survey [16].

The election day voting centers described in Chapter 2 will naturally have larger numbers of votes cast than traditional small precincts. Voting machines could easily be grouped into subsets that would have separate Auditorium networks and separate homomorphic tallies. Similarly, over a multi-day early voting period, each day could be treated distinctly.

7.4.1 Network load

The most burdensome part of the Auditorium network is its all-to-all connectivity graph. Each message a node wishes to broadcast in the Auditorium results in roughly n^2 messages on the network. More exactly, the originating node sends the message to $n - 1$ neighbors, each of which will forward the same message to their other neighbors, resulting in $n - 2$ more messages for each. Each of these messages should be old news to its recipient by this point, and so the flood stops here. The total number of messages sent is therefore $n + (n - 1)(n - 2)$; we round up to n^2 (100) for our hypothetical polling place.

Assuming ballots are cast every 3 minutes on each of 10 machines, we have 200 ballots per hour, or 600 messages per hour (since each cast ballot is the result of an **authorized-to-cast, cast-ballot, received-ballot** message exchange). Since each Auditorium broadcast results in roughly 100 actual message transmissions, we now have 60,000 messages per hour, or about 17 messages per second. Add to this the periodic heartbeat messages, which (if issued every 5 minutes by each node) bring our total to 20 messages/sec.

Most messages are on the order of 1 KiB (see Chapter 7 for concrete figures including the cryptographic measures introduced in Chapter 4). This corresponds to a maximum cross-sectional network bandwidth requirement of roughly 164 kbps, which a 10base-T Ethernet hub can handle with plenty of headroom for other incidental messages (node join, polls closing, etc.) in the Auditorium.

Not every message fits in one or two packets, however; the **authorized-to-cast** message, in particular, contains a complete ballot definition, which in our prototypes (including all pre-rendered UI elements) is roughly a megabyte in size. This traffic now dominates our bandwidth calculations; 200 ballots per hour result in up to 20,000 **cast-ballot** messages per hour, or about 6 per second, which is about 50 megabits and requires a faster network.

The following optimization mitigates this problem. Because the largest part of a ballot definition is its collection of pre-rendered images (see Chapter 6), these can be distributed to machines ahead of time. An **authorized-to-cast** message would then include only the logical ballot definition itself, which in turn references these images. The integrity of image files can be verified by including the cryptographic hash of each image in the ballot definition (or in the image's filename itself, making

images self-certifying). In this way we can be sure that even though a ballot definition does not carry its own images, the correct image will be displayed. A further step is to distribute the entire ballot (a zip archive) and reference that ballot in an `authorized-to-cast` simply by name (and cryptographic hash). With this final optimization, ballot authorization messages are brought down to a very small fixed size (approximately 1 KiB like other Auditorium messages).

7.4.2 Bandwidth requirements of the challenge scheme

An important feature of the challenge system described in Chapter 4 is the ability of outside parties to see a polling place's Auditorium log data in real time so that they may assist in the verification process by decrypting challenge responses.

Were the third-party challenge verifier to reside on the LAN, the problem of supplying it with log data reduces to the problem of distributing messages to any system in the Auditorium network, which is shown in Section 7.4.1 to be entirely tractable for an Ethernet network.

Most likely, the verifier will reside offsite; connectivity may be at DSL speeds, but a more conservative (and economically practical) assumption is that the polling place may only be connected to the verifier over a telephone line and at modem speeds. Therefore, an analysis of the feasibility of transmitting log messages in such a bandwidth-constrained environment is warranted.

A single voter's interaction with the VOTEBOX booth results in the following sequence of log messages broadcast in Auditorium:

1. an `authorized-to-cast` message from the supervisor to the booth (shortly after the voter enters the polling place);
2. a `commit-ballot` message broadcast by the booth after the voter is done voting;
3. a `ballot-received` message from the supervisor, acknowledging receipt of the ballot;
4. either a `cast-committed-ballot` (if the voter casts the ballot as usual) or `challenge-committed-ballot` (if the voter challenges);
5. a `ballot-counted` (if the ballot was cast) or `foo` (if challenged) from the supervisor, which effectively allows the machine to release its state and wait for the next authorization.

SEQUENCE	MESSAGE	KiB (CAST)	KiB (CHALLENGE)
1	authorized-to-cast	1	
2	commit-ballot	$0.5 + n$	
3	ballot-received	1	
4	cast-committed-ballot	1	
	challenge		$0.5 + 0.5 n$
5	ballot-counted	1	
	challenge-response		1
TOTAL		$4.5 + n$	$5 + 1.5 n$

Table 7.2: Bandwidth of Auditorium messages involved in a voting session. Figures are approximate; Auditorium messages contain about 0.5 KiB of overhead and typically include another 0.5 KiB of voting-specific data. The exceptions are `commit-ballot`, which includes two large numbers (about 1 KiB total) per counter, and `challenge`, which includes one large number (about 0.5 KiB) per counter.

The smallest Auditorium message, including a certificate (with 1024-bit cryptographic key), digital signature, and other basic metadata, is 542 bytes. (See Appendix A for documentation of the Auditorium wire protocol.) All Auditorium messages used in VOTEBX include at least a message name, some number of preceding message pointers, and so forth; most still fit in 1 KiB. This includes the `authorized-to-cast` message, provided that ballot contents are not distributed anew with each authorization, as ballots with prerendered graphics can be quite large. Such an optimization, noted in Section 7.4.1, is straightforward and does not impact the semantics of the logs as long as ballots are referenced by cryptographic hash.

Encrypted ballots (`commit-ballot` messages) contain about 1 KiB of data per counter (each being an El Gamal ciphertext, represented as a pair of large integers in decimal format, on the order of 500 bytes apiece). Challenge responses include a single large integer (r) per counter, for a total of $\frac{n}{2}$ KiB. Assuming 30 selectable elements are on the ballot, both commit and cast messages are 13 KiB while challenge response messages are 7 KiB. An acknowledgment is 1 KiB. The network overhead of a single vote is summarized in Table 7.2.

Considering the following scenario:

- The ballot is of moderately large size, containing 30 contests.
- 20 voters, the maximum number of VOTEBX booths considered in Section 7.4.1, are voting

simultaneously.

- The polling place is connected to a challenge center via a 56K modem (about 5 KiB/s throughput).⁸

The challenger must ask the machine to commit to a vote, wait for the verification host to receive the commitment, then ask the machine to challenge the vote. (The voter *must* wait for proof of the booth's commitment before challenging the system, to ensure that the commitment is made without knowledge of the impending challenge.) To make this scenario most extreme, we assume the challenger is “last in line:” his VOTEBOX's commitment message must wait behind vote data for 19 other voters, totaling about 650 KiB of queued log data (approximately 130 seconds). Once this happens, the subsequent messages (the challenge and challenge response) happen quickly. The total delay experienced by the challenger in this case is between two and three minutes.

7.4.3 Storage

Storage is the least expensive and most plentiful resource in a VOTEBOX; assuming our worst-case hypothesis from Section 7.4.1, a machine might receive 5 messages per second, of which a third contain large ballot files, but because most of these are duplicates (thanks to flooding), we need store very few. We rely on our original estimate of 200 ballots per hour, or 2400 per day (assuming the polling place is open for 12 hours), which means 4800 small messages (cast-ballot and received-ballot; heartbeats are on this order of magnitude as well) and another 2400 large messages (authorized-to-cast).

The large messages dwarf the smaller ones, so we have roughly 2400 MB of data to store, supportable even with solid-state flash memory. With a hard drive we have the luxury of preserving this data forever; a VOTEBOX with a hard drive needs no “clear” function, making it still harder to accidentally destroy election records. A caching strategy for ballot definitions, as described in Section 7.4.1, would reduce even this requirement considerably; storing only small messages now, we need on the order of a few megabytes, allowing even flash memory to operate without needing a “clear” function.

⁸This assumes a relatively noise-free line (allowing full 56K negotiation) and a direct connection protocol (that is, roughly equivalent to dumping Auditorium data over a serial port at an equivalent speed).

7.4.4 CPU demands of encryption

Current electronic voting systems are, at heart, general-purpose computers using commodity processors. Some are quite powerful:

- The Premier/Diebold AccuVote TS used a 133 MHz Hitachi SuperH SH7709A [35]; the AccuVote TSx uses a 400 MHz PXA255 (an ARM architecture chip in Intel's, now Marvell's, XScale line). [15]
- The Sequoia AVC Edge contains a 300 MHz National Semiconductor Geode (an x86 architecture embedded chip). [15]

On the other hand, voting systems using custom embedded operating systems (and presenting substantially simpler user experiences) can get by with far more modest hardware:

- The ES&S iVotronic uses a 25 MHz Intel 386EX. [15]
- The Hart InterCivic eSlate uses a 90 MHz Freescale Semiconductor ColdFire MCF5307 (based on the Motorola 68000 architecture). [44]

The VOTEBOX is designed to run atop this sort of general-purpose computer hardware, so the computational effort needed to participate in Auditorium must be within the bounds of such a system.

Three types of cryptographic operations commonly occur in VOTEBOX:

- Cryptographic hash computation (SHA-1);
- RSA signature operations; and
- El Gamal encryption operations.

We used OpenSSL's built-in speed microbenchmark tool on an unloaded 300 MHz Pentium II with 256 MB RAM (OpenSSL 0.9.7e compiled with `-march=i686`, Linux kernel 2.6.8). This is, by modern standards, a dinosaur; indeed, many common mobile phones today deliver more horsepower. It is used here as an extreme case, a model of the kind of low-end, commodity hardware that might be used for inexpensive voting systems.

This example test machine can perform 796 1024-bit RSA verifications and 42 signatures per second; it hashes 20 MB of data per second with SHA-1. Such performance is more than sufficient for the workload described here. (A deployment of Auditorium requiring substantially greater cryptographic performance could be achieved straightforwardly by provisioning hardware adequate to the task.)

In practice, hardware used for testing and demonstration—including several Macintosh computers and PC laptops and desktops, ranging from 800 MHz to several gigahertz—was mostly idle while running VOTEBOT.

Our El Gamal cryptosystem does not similarly benefit from a well-optimized external library; VOTEBOT includes a pure-Java implementation of the El Gamal cryptosystem, relying on Java's BigInteger class. To characterize the CPU demands of El Gamal encryption operations, we benchmarked the encryption of a reference 30 candidate ballot. On a Pentium M 1.8 GHz laptop it took 10.29 CPU seconds, and on an Opteron 2.6 GHz server it took 2.34 CPU seconds.

We also benchmarked the decryption, using the r -values generated by the encryption function (simulating the work of a verification machine in the immediate ballot challenge protocol). On the laptop, this decryption took 5.18 CPU seconds, and on the server it took 1.27 CPU seconds. A third party challenge center supervising several hundred precincts at once (Harris County, which contains Houston, has about 800 precincts), may need to handle 50 ballots per second (using the estimated ballot casting rate and number of voting machines from Section 7.4.1). If every single ballot were a challenge, 40 Opteron CPUs would therefore need to be dedicated to the task of decrypting them. A more reasonable challenge rate, for example 1% of ballots cast, reduces the problem to a more manageable size (the example server CPU would be roughly half idle).

It is important to note that decryption is slightly more complex than encryption in our cryptosystem. To make our encryption function *additively* homomorphic, we exponentiate a group member (called f in Equation 4.1) by the plaintext counter (called c in Equation 4.1). (The result is that when this value is multiplied, the original counter gets added “in the exponent.”) Because discrete log is a hard problem, this exponentiation cannot be reversed. Instead, our implementation stores a precomputed table of encryptions of low counter values. We assume that, in real elections, these counters

will never be above some reasonable threshold (we chose 20,000). Supporting counters larger than our precomputed table would require a very expensive search for the proper value.

This is never an issue in practice, since individual ballots only ever encrypt the values 0 and 1, and there will never be more than a few thousand votes per day in a given precinct. While there may be a substantially larger number of votes across a large city, the election official only needs to perform the homomorphic addition and decryption on a precinct-by-precinct basis. (As noted in Section 7.4, larger precincts such as vote centers can be subdivided into Auditorium networks of a manageable size.) This also allows election officials to derive per-precinct subtotals, which are customarily reported today and are not considered to violate voter privacy. Final election-night tallies are computed by adding the plaintext sums from each precinct.

CHAPTER 8

EXTENSION: REMOTE VOTING

8.1 Introduction

Never. This, the answer given by e-voting security researchers when asked when we will be able to vote in national elections over the Internet, is unsatisfying to many because of the tremendous convenience it would seem to afford (see, e.g., Alvarez and Hall [8]). The number of endeavors, from personal entertainment to securities trading, that have been profitably brought online would imply that the Internet can improve any task that does not absolutely require one to be physically present.

Voting, unfortunately, requires absolute trust in two factors that cannot be adequately controlled in the residential Internet scenario: *environment* and *equipment*. The voter's PC may be compromised; the voter may be coerced. It is not the only such task; academic testing, for example, requires a testing environment free of distraction, collusion, and unauthorized assistance.

Participating in national elections from the comfort of one's home computer may never be practical or secure, but *remote* voting can be both. Voters in many jurisdictions are currently permitted to cast provisional ballots in situations where their eligibility to vote is in doubt; the voter's identification is submitted along with the sealed ballot for consideration by elections officials. Postal voting (or "vote-by-mail") functions similarly. Each of these schemes trades a certain amount of anonymity for the ability to determine the eligibility of a prospective voter.

These techniques inspire the following vision of practical electronic remote voting. A DRE that encrypts individual ballots provides the sealed ballot described above; when digitally signed along with plaintext attesting the identity of the voter, it becomes an electronic replica of the conventional

provisional ballot, albeit one that can travel faster and more safely than a postal envelope.

Voters far from their home precincts could therefore visit a *remote voting center*: a facility maintained and supervised by government officials, perhaps in foreign embassies or in controlled areas on military bases and ships. The establishment of such a “remote precinct” for voters far from home dates at least to 1864, in which soldiers fighting in the American Civil War voted in temporary battlefield polling places [100].

The version of the remote voting center described here consists of one or more electronic voting booths and a registration system. Voters present their personal identification, and are then directed to cast a ballot in a private electronic voting booth with the proper local ballot (provided in advance by the election director of the voter’s home precinct). The cast ballot is encrypted and returned to the registration system, which then in turn wraps the ballot ciphertext in the voter’s identifying information. This might include a scanned signature or ID card or even a digital photograph of the voter taken at the time of voting. This double enclosure is then digitally signed by the voting center and posted on a public “bulletin board” where it may be examined and canvassed by the voter’s home election officials. Once the election officials have determined that the ballot was cast properly (e.g., the voter’s identification matched up with records on file and the proper ballot definition was used), then they can approve the still-encrypted ballot for inclusion in the final tally.

In this chapter I review the procedures currently in place for postal and provisional balloting (Section 8.2), giving special attention to the security guarantees made to the voter for these (Section 8.3), and finishing with a sketch of remote voting using the `VOTEBOX` platform (Section 8.4).

8.2 Provisional and postal voting

Postal voting is used widely in the U.S. and is growing in popularity. The state of Oregon, for example, votes exclusively by mail. Many states offer “no fault” postal voting; voters may declare their desire to vote by mail without requiring any reason. In California, voters may declare their desire to vote exclusively by mail, and need never again cast ballots in person.

A month in advance of the proper election date, ballots are mailed to these voters, giving the voter time to cast the ballot or request an alternative ballot if the original is lost or spoiled. Completed

ballots are placed in an opaque return envelope. The back of this envelope has designated areas for the voter to inscribe her personal identifying information, including her signature. A paper flap (or, in some cases, another enclosing envelope) conceals this personal information while the ballot is in transit.

When envelopes arrive at the elections office, they are counted and stored. Each envelope's signature and personal information is verified by hand against the voter's registration data. If an envelope is rejected, election officials may then attempt to contact the voter to offer an additional opportunity to cast a vote, assuming the election is still ongoing. Ballots that are determined to be legitimate are then removed from their envelopes and stored as any other ballot might be stored. These ballots can then be tabulated using the same optical scan machinery that can be used for paper ballots cast in traditional precincts.

Provisional voting, required as part of the 2002 Help America Vote Act, is semantically similar to postal voting. Provisional voting occurs when a voter arrives at what she believes to be the proper precinct only to discover that she is absent from the precinct's registry of voters. At that point, she may conclude that there was an error and declare the desire to cast a vote, regardless.

Procedures for this vary from voting system to voting system. One common solution is that the voter is handed a paper ballot along with an envelope. The paper ballot is filled out, as normal. The envelope, much like a postal voting envelope, contains information about the voter's identity along with why he or she claims the right to cast a vote in this particular precinct. Some DRE voting systems offer similar functionality, tagging provisional votes with an identifier of some kind that corresponds to paper records describing the voter's situation.

Provisional votes are generally not tallied until a recount occurs or if the number of provisional votes is large enough to impact the election's outcome. At this stage, election officials hold a public hearing to individually discuss each provisional voter and determine whether his or her vote will be counted. Once the envelope has been validated, the inner ballot can be removed and tabulated.

8.3 Security and privacy of remote voting

8.3.1 Conventional approaches

Voter anonymity is necessarily harder to safeguard when the voter's name, address, and signature accompany each ballot. Present-day provisional and postal voting attempt to preserve privacy through a combination of technology (ballots are enclosed in opaque envelopes) and procedure (envelopes are only opened if eligible, and once validated, a ballot is separated from its envelope).

Postal voting, however, suffers from several obvious problems. The postal mail channel is slow and not sufficiently reliable, particularly when delivering mail overseas. Furthermore, there are a wide variety of opportunities for election fraud with postal voting, ranging from outright bribery and coercion (i.e., selling unvoted ballots) to attacks upon and within the postal system (e.g., postal workers destroying or tampering with ballots). While some voters may detect that their ballots failed to arrive at their destination, it would be difficult to automatically detect and correct such errors. In cases where the postal delivery channel is too slow or too lossy, multiple round-trips with the voter are likely infeasible in the time allotted for voting.

Provisional voting, when performed inside a properly supervised voting location, is more robust against bribery and coercion, since the vote will have been cast in the privacy of a voting booth. Likewise, there are fewer concerns about loss or damage to votes while in transit. Nonetheless, as with postal voting, the ties between the voter's identity and ballot allow subsequent opportunities for fraud, whether the provisional vote is cast on paper or with current-generation DRE systems. Election officials, therefore, must be trusted to properly manage the process to preserve voters' privacy.

Both postal and provisional voting share the property that a variety of attacks can be *detected* even when they cannot necessarily be *corrected*. Voters can detect whether their ballots were received and whether they were tabulated. They cannot learn whether their ballots were tabulated accurately.

8.3.2 Goals for a networked replacement

A remote voting system design that proposes replaces postal mail with Internet transmission must retain comparable security and privacy semantics to postal or provisional voting. While the ballot

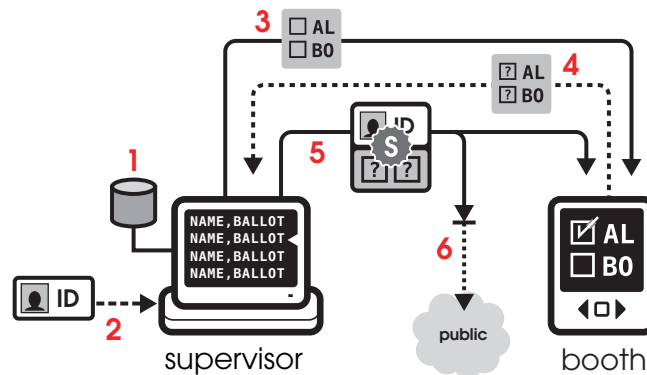


Figure 8.1: Voting with RemoteBox. A database (1) is furnished in advance, containing a blank ballot design for each voter allowed to vote at this location. On election day, the voter presents identification (2) which is used to select the correct ballot for voting. As in a conventional polling place, the blank ballot is sent to a VOTEBOX (3) for voter input, and returned in the form of an encrypted cast ballot (4). The supervisor combines the result with the voter’s identification and signs it (5), broadcasting it to the polling place for storage as well as on a one-way channel to a public medium (6).

should be accompanied by information identifying the voter so that only eligible remote votes are counted, there must still be some sort of opaque envelope; the voter’s choices must be concealed until eligibility is determined, and then separated from the voter’s identity before tabulation.

Problems should also be detectable (even if they cannot be immediately corrected) in such a system. For example, as Internet hosts are indisputably vulnerable to denial of service attacks, the voter must still be allowed to cast a ballot regardless of whether or not the election authority can be reached from the polling place. That is, despite the supposition of a network connection, the remote polling place cannot use “online” methods that require constant uptime of that connection. Finally, such a design should improve on postal voting by providing a voting environment that resists voter coercion and fraud.

8.4 RemoteBox: connecting remote precincts over the net

8.4.1 Remote electronic voting

The proposed remote voting environment is built atop the VOTEBOX design of Chapter 3. The “RemoteBox” remote polling place adds the following to a VOTEBOX polling place:

- A **remote polling place**, maintained and monitored by trusted / non-partisan government officials. Such a facility might exist in embassies, consulates, and military bases—anywhere a large population of remote voters may be served.
- A database of **eligible remote voters**, mapping name and home precinct information to the correct blank ballot design for that voter. (Jurisdictions wishing to allow their voters to cast remote electronic ballots must furnish this information in advance.)
- A requirement that the voter **present identification** on election day: a government-issued ID or voter registration card, plus an interactive authenticator like a handwritten signature. Just as with a postal or provisional ballot, the voter must be identified so that he or she may be given the correct ballot, and so that election officials can decide whether the voter’s cast ballot should be counted.
- The notion of a **provisional electronic ballot**, which is a signed enclosure certifying the identity of the voter and her encrypted vote. This is an analogue of the conventional provisional ballot envelope (identifying the voter outside and sealing her choices inside) described in Section 8.2.
- A one-way channel to a **public medium**, for posting provisional electronic ballots. This could be an online channel such as an Internet link (perhaps on the other side of a data diode [61]) or an offline one such as a CD-ROM burner.

Figure 8.1 illustrates how these components fit together on election day.

8.4.2 Ballot definitions

An obvious complexity in this system lies in managing the *ballot definitions*, which will vary widely from county to county and state to state. If there were a single, standardized, national voting system, particularly based on pre-rendered ballots, then these ballot definitions might also be collected by a centralized organization within the federal government. State- and locality-specific issues (e.g., Texas requires a “straight ticket” voting option while California forbids it) would be encoded in the ballot definitions, requiring the remote voting machines to be sufficiently generic to accommodate any voter from any jurisdiction. Current DRE systems’ ballot preparation tools could be augmented to output a standardized description of the ballot which could then be processed independently.

8.4.3 Cryptographic and pragmatic details

Key management. As described in Chapter 3, VOTEBOX requires that every piece of voting equipment (booths and supervisors) to have its own local key material for digital signatures. Moreover, each jurisdiction’s election administration office (county clerk, etc.) must have individual public keys such that ballots cast remotely may be encrypted for their eyes only. This problem is similar in scope to the issues surrounding ballot definitions (described in the prior section). Again, assuming the existence of a centralized organization within the federal government, this key material could be collected and redistributed in advance of elections. Ballot definitions could likewise be centrally collected and disseminated. Each ballot definition would include the appropriate public key to use when encrypting votes cast with that ballot.

Bulletin boards. A standard feature of many cryptographic voting protocols is the concept of a bulletin board where ballots are posted for all the world to see. This could serve as a mechanism for disseminating the results from remote voting precincts back to their proper home for tabulation. With proper key management and ballot definition distribution, performed in advance of the election, local election officials should easily be able to identify ballots on the bulletin board which are intended for their local consumption. These ballots would be encrypted with local election officials’ public keys and signed with the keys of the remote voting system. The entire bulletin board

from each remote precinct could then be signed by the remote precinct itself, protecting the bulletin board against tampering.

Because the bulletin board publishes encrypted ballots alongside the plaintext identity of the voter, there may be some danger of very long-term anonymity compromises due to hypothetical future computational advances or other weaknesses in the encryption used. Any mitigation of this risk is going to require either weakening the binding between a voter and his or her encrypted vote, or making the channel for distributing these votes less observable to the public than a bulletin board. For example, the bulletin board could hold only statistically-hiding vote commitments, rather than the encrypted ballots themselves. In such a scheme the actual ballots must be transmitted privately and verified against the public bulletin board (as in, e.g., Moran and Naor [72]).

Networks. Ultimately, the ballots (on a bulletin board or otherwise) must be transmitted from remote polling places to election officials. As a real-time feed is unnecessary (and possibly infeasible for some remote locations), ballots may be batched and sent infrequently, perhaps at the end of each day.

This allows some flexibility in how exactly to transmit ballot data. For example, in order to isolate the remote precinct from the Internet, the supervisor console might burn a CD-ROM. This could be transmitted via an overnight courier or hand-carried to a computer connected to any sort of network, whether public or private. All the remote results could be aggregated (but not tabulated) by the same centralized federal agency that coordinated the distribution of cryptographic keys and ballot definitions.

If this agency should suffer sustained attacks on its Internet connection, then alternate procedures could be used. All of the election results could be disseminated through slower means (mailing CDs, etc.). All that matters is that the various cryptographic signatures are properly verified, which can be done both by local election officials and by the remote voting center's officials.

Various attacks. A voter with access to multiple remote voting centers (or, perhaps, a coalition of attackers using the stolen identity of one valid voter) could use the system described thus far to cast one vote per voting center. This would not necessarily be detected during the voting day. Nonetheless,

each encrypted vote would be contained in a public envelope with the voter's identifying information present. Election authorities could certainly detect multiple votes having been cast, exactly as they can in the case of postal or provisional voting. It then becomes a policy problem to determine which vote should be counted and whether a crime has been committed. Alternatively, voters could be required to declare, in advance, which remote voting center they intend to use. When a voter shows up at the proper remote voting center, his or her name is present and the vote proceeds normally. At other remote voting centers, the voter would be absent from the database and could then only vote provisionally.

8.5 Summary

The remote polling place is a model for networked remote voting that brings the benefits of DRE voting (convenience, speed, fault-tolerance) to provisional and postal voting. The security and privacy guarantees of these conventional remote voting methods are met or exceeded by this approach. As shown in Section 8.4, the VOTEBX system design can be straightforwardly extended to accommodate this voting model by enclosing anonymized, encrypted ballots in a public wrapper identifying the voter. A similar transformation (comprising a remote polling place and double-enclosure provisional ballots) should be applicable to any DRE-style voting system, provided that it is engineered (or re-engineered) with the necessary properties from VOTEBX: it must accommodate a potentially large number of ballot designs (perhaps by loading them on-the-fly per voter), and it must provide the essential “opaque envelope” by encrypting each individual cast ballot.

CHAPTER 9

CONCLUSION

In this thesis I have shown how the VOTEBOX system design is a response to threats, real and hypothesized, against the trustworthiness of electronic voting. Recognizing that voters prefer a DRE-style system, and that such systems have advantages over non-electronic voting systems, I and my colleagues in the Computer Security Lab at Rice University created VOTEBOX: a system that superficially resembles today's flawed electronic voting machines, but is built on sound techniques from distributed systems, cryptography, and e-voting security research.

VOTEBOX uses a novel networking infrastructure called Auditorium to coordinate election-day operations and replicate ballots and auditing records. The resulting logs, which contain powerful proofs of integrity and order, withstand scrutiny even when machines are damaged or faulty and when data is lost or incomplete. This work spurred creation of the Querifier log analyzer, the first tool designed to evaluate arbitrary predicates on secure logs of this form. Electronic voting over Auditorium is an excellent substitute for today's provisional and postal voting, offering pragmatic benefits and additional security properties.

To address the vexing problem of software independence, VOTEBOX adapts and enhances work by Benaloh on user-initiated auditing. The ballot challenge scheme allows any voter to force any VOTEBOX to prove its correctness and honesty, on election day and in the polling place, in such a way that the machine cannot guess that it may be under test. By combining the challenge system with Auditorium, VOTEBOX offers would-be challengers the ability to make use of an offsite third party of their choosing to assist with the decryption process needed to verify the machine's challenge response, thus overcoming the awkwardness of the corresponding step in Benaloh's original proposal.

Borrowing a technique from Yee, VOTEBOX moves inessential graphics code out of the critical voting booth software, resulting in a smaller (and more easily analyzed) software artifact. The pre-rendered ballots used with this technique can be created—and audited—long before election day. Inspired by work by Sastry, VOTEBOX destroys most of its data structures between voters to avoid accidental data leakage from one session to the next.

Some of the elements of VOTEBOX are not novel in themselves; the system borrows sophisticated techniques, where appropriate, from other researchers in the field. Others (particularly Auditorium) are entirely new inventions, inspired by the problems identified by other scholars as well as my own observations in the field. Additionally, VOTEBOX is a novel demonstration of the way in which these various approaches may be integrated in a coherent whole to achieve the project's overall security goals.

The future of VOTEBOX is multi-dimensional. Along one axis, it continues to demonstrate value as a research platform. As described above, VOTEBOX brings computer science techniques (distributed systems; secure logs; cryptography) to bear on certain aspects of the voting problem: reliability, integrity and audit, and verification of correct operation. The solutions used in VOTEBOX have been carefully chosen so as not to erode the voter's privacy, but it is important to note that these measures do little to *increase* privacy protections. A malicious VOTEBOX cannot destroy or alter auditing data without detection, nor can it satisfy a cryptographic challenge with an incorrect ballot. But it could leak the voter's choices in any number of ways: by choosing a non-random r (allowing an untrusted party to decrypt votes), by saving r values somewhere, or even by saving vote plaintext somewhere. It could distribute vote plaintexts on the network, or over a wireless channel.

However, these problems exist for any voting technology currently proposed or in use, including paper ballots (e.g., malicious optical scanners, or hidden cameras in the polling place). The voter can easily violate his own privacy by bringing a cameraphone into the booth with him. Privacy, therefore, is a large and open problem. Improving the state of the art here is a laudable goal that is also beyond the scope of this thesis.

Other scholars continue to use VOTEBOX in their investigations. Human factors research, into such questions as “which navigation schemes are best?” (viz., current work by Greene [42]) and “do

voters notice malfunctions or malice?” (as in a thesis by Everett [34]) are best conducted using a real voting system such as VOTEBOX.

Additional security features, such as non-interactive zero-knowledge proofs (NIZK) and trusted computing techniques (TPM), are the subject of work by other researchers in our lab, and future work of this sort may involve actually developing a minimal Java runtime (of the sort used today in some embedded contexts) upon which to deploy a very compact VOTEBOX. Pragmatic additions to the system—such as the integration of additional user input devices, ballot printers, voter registration mechanisms, and so on—may not be of interest from a security or reliability standpoint, but would flesh out the system and start to move it out of the laboratory and toward a realizable *product*.

This, then, is the second direction along which VOTEBOX may begin to have impact. Techniques, designs, and even code arising from this work can find use in commercial voting systems. I have taken care to design a system that is practical for new implementations on very modest computer hardware; crucial elements of the system architecture may even be added on to existing systems to improve their security properties. For example, the techniques presented in Chapter 3 could be applied to non-DRE electronic voting equipment; an Auditorium network of optical ballot scanners would bring robustness and security to paper ballots.

Beyond electronic voting, I believe some of these inventions can be applied to other problem domains. The Auditorium network, in particular, should be useful in systems that must maintain believable records that are tamper-evident, provably time-stamped, and recoverable in case of failure or malice. This might include games that require defense against cheating; distributed collaborative document editing using Auditorium’s provable ordering semantics; and Internet-scale messaging systems (such as email, instant messaging) and publishing systems (e.g., Facebook and Twitter, the subject of recently published work [89]).

APPENDIX A

AUDITORIUM PROTOCOL FOR VOTING

A.1 Auditorium messages

The Auditorium system described in Chapter 3 is a self-organizing, decentralized network structure designed to facilitate the creation and exchange of secure log data. Although developed for VOTEBOX, it is able to support general peer-to-peer applications that involve secure logging or timeline entanglement.

In this appendix I detail the data structures that constitute messages in Auditorium and VOTEBOX. As noted in Chapter 7, the syntax used by Auditorium for log messages—whether on the wire or in persistent storage—is based on S-expressions, the recursive prefix-notation *symbolic expressions* developed for LISP by McCarthy [70]. The specific representation chosen for Auditorium is borrowed from Rivest [77], chosen (as outlined more fully in Section 7.2.3) for its unambiguous and convenient representation in cryptographic and networking contexts (Rivest’s “canonical” encoding).

The following sections catalog the specific messages used in VOTEBOX; they form the protocol used for network communication as well as the storage format for secure logs in the system. They are shown not in canonical encoding but in a more friendly representation familiar to users of LISP (referred to by Rivest as “advanced” encoding).

A.1.1 Data structures

The following data structures can be found inside auditorium messages.

Message Pointer

A *message pointer* is a reference to a particular message.

```
(ptr [node-id] [sequence-number] [hash])
```

- **node-id:** This machine serial number uniquely identifies the machine which sent the message.
- **sequence-number:** This message number is an index into the sequence of messages sent by the sending machine.
- **hash:** This is the SHA-1 hash of the message in its canonical S-expression format as it was placed on the wire.

Therefore, the <ptr> structure uniquely describes a single message from an Auditorium node, including the message's hash (for integrity checking), and implicitly attesting to the state and integrity of that node's entire log (thanks to hash chaining).

Host Pointer

A *host pointer* is a reference to a particular host on the network.

```
(host [node-id] [ip] [port])
```

- **node-id:** This machine serial number uniquely identifies the machine which sent the message.
- **ip:** This is the IP address of the referenced host in dotted-decimal format
- **port:** This is the port that the host is listening for incoming connections on. It is an integer formatted as a string.

Certificates and Signatures

A *certificate* is a key signed by an identity:

```
(cert <signature>)
```

A *key* combines an RSA modulus and exponent and contains an *id* (such as the ID of a VOTEBOX booth or certificate authority) and an annotation (arbitrary text for the convenience of humans examining the file, such as “certificate authority” or “booth”):

```
(key [id] [annotation] [mod] [exp])
```

A *signature* is:

```
(signature [id] [sigdata] {payload})
```

Therefore, a certificate for `votebox-5` signed by the certificate authority (with key ID `ca`) has the form:

```
(cert (signature ca
      [sigdata]
      (key votebox-5 booth [mod] [exp])))
```

A.1.2 Messages

All auditorium messages have the following format:

```
([name] <host-pointer> [sequence-number] {payload})
```

- **name:** Identifies the type of Auditorium message. May be one of:
 - `discover`
 - `discover-reply`
 - `join`
 - `join-reply`
 - `announce`
- **host:** A reference to the sender of the message (see “Host pointer” above).
- **sequence-number:** A unique identifier (across only messages from this sender) for the message.
- **payload:** The contents of the message.

Discover

This message is broadcast by a host who seeks information about other auditorium hosts nearby. This is a special message that may be broadcast on a local network segment via UDP if the sender does not

know any IP addresses to connect to. The payload is a host pointer to the host that the responding host should send its discover-reply to.

Example:

```
(discover <ptr> <seqno> <host>)
```

Discover Reply

This message is sent as a reply to a host who broadcast a discover message. The payload consists of all known hosts. Even if discover is sent via UDP, the reply—as well as all subsequent Auditorium messages—should be sent via a TCP stream.

Example:

```
(discover-reply <ptr> <seqno> (<host> <host> ... <host>))
```

Join

This message is sent by a host who wants to start a link with another host. There is no payload.

Example:

```
(join <ptr> <seqno>)
```

Join Reply

This message is sent in reply to a join in the event that the joiner is considered acceptable by the receiver of a join. In the event that the joiner is not considered acceptable, the socket is simply closed by the receiver of the join. The payload contains a list of message pointers, indicating messages that have been seen on the network but not yet referenced by a message; the purpose is to give the newly joined node an opportunity to create its first message in context of the DAG of time already in progress. (Alternatively, all new nodes would be forced to either wait to hear a message in order to “find their place” in the timeline, or simply broadcast an initial message with no predecessor whatsoever.)

Example:

```
(join-reply <ptr> <seqno> (<ptr> <ptr> ... <ptr>))
```

Announce

The announce message is the workhorse of Auditorium; unlike the messages described thus far, which are concerned with the construction of the network structure, announce is used for communication between nodes. The payload is variable, and any S-expression useful for the application (typically a signed message) may be inserted here.

For convenience, we define a signed-message structure as follows:

```
(signed-message <certificate> <signature>)
```

The <signature> is as defined above. Such a signed-message structure would be used in the payload of an announce message.

We also add the succeeds structure, allowing us to combine a new message text (the payload) with a number of predecessor messages (the list of message pointers):

```
(succeeds <list-of-ptrs> <payload>)
```

An extended, concrete example, involving announce as well as the signed-message, succeeds, and ptr data structures (semicolons are used to set off comments that are ignored by the parser):

```
(announce
  (host votebox-5 1.2.3.4 5555) ; the sender in <hostptr> format
  1 ; sequence number for the message
  (signed-message ; the payload is signed
    (cert ; certificate of the sender/signer
      (signature ca [sig-data] (key votebox-5 booth [mod] [exponent])))
    (signature ; the signature itself
      votebox-5 ; keyid of the signer
      [sigdata]
      (succeeds
        ( ; list of prior messages
          (ptr votebox-1 38 [SHA1-hash])
          (ptr votebox-0 15 [SHA1-hash])
        )
        "hello world")))) ; finally, the message contents
```

A.2 Voting messages

The foregoing messages form a protocol that is sufficient to start an Auditorium network and exchange information among nodes. In order to conduct an election using a number of VOTEBOX

booths and supervisors, a subprotocol is required. The following messages are to be understood as the innermost payload of the announced signed message outlined above.

A.2.1 Sent by supervisor

- (**polls-open** *local-timestamp keyword*)
 - The *keyword* is given to the poll workers on the morning of the election, in order to guarantee the results were not fabricated in advance.
 - The *local-timestamp* may be unnecessary here.
 - State change: Booths become active, and the Supervisor can begin transmitting **authorized-to-cast** messages.
- (**authorized-to-cast** *node-id authorization-code ballot*)
 - The *authorization-code* is a stream of random bytes determined by the supervisor. (It cannot be guaranteed that this will be unique to the election or to any machine.)
 - State change: The booth's state changes to "In Use", and the booth loads the ballot and prepares to accept user input.
- (**override-cast** *node-id authorization-code*)
 - State change: The Booth shows a message that the ballot is about to be overridden and cast, and asks for confirmation.
- (**override-cancel** *node-id authorization-code*)
 - State change: The Booth shows a message that the ballot is about to be overridden and cancelled, and asks for confirmation.
- (**ballot-received** *node-id authorization-code*)
 - reply to **cast-ballot** (or **commit-ballot** when the challenge system is in use)

- State change: The Booth informs the voter that the ballot has been received. If the challenge system (Chapter 4) is not in use, this is the end of the voting session; the booth changes to the “Ready” state, and the *authorization-code* is de-authorized.
- (**ballot-counted** *node-id authorization-code*)
 - reply to **cast-committed-ballot**, used in conjunction with the challenge system when a ballot is not challenged but is instead confirmed by the voter.
 - State change: The Booth informs the voter that the ballot has been successfully cast; the voter’s session is over. The machine changes to “Ready”, and the *authorization-code* is de-authorized.
- (**challenge-response** *node-id authorization-code*)
 - reply to **challenge**, used during a ballot challenge.
 - State change: The Booth informs the voter that the ballot challenge has been successfully issued; the voting session is over. The machine changes to “Ready”, and the *authorization-code* is de-authorized.
- (**supervisor** *local-timestamp supervisor-status*)
 - *supervisor-status* can be active or inactive
 - This message is sent once when the supervisor connects to Auditorium, and periodically as a status message.
- (**assign-label** *node-id new-label*)
 - The supervisor shows the machine’s new label. Also, the booth should remember its label so that if a new supervisor comes on, it doesn’t need to relabel every machine.
- (**polls-closed** *local-timestamp*)
 - State change: Booths go into inactive mode, and the Supervisor can no longer authorize voters. Tallying functionality may become available.

- (**activated** ((**status**)*))
 - This is sent when a supervisor console is activated by a user (user presses the big “Activate This Console” button that actually turns on the UI). Why do we need an “active” supervisor? Because of automatic booth labelling. If the super is supposed to automatically assign a label to every booth as the booth appears, we can’t have backup supervisors competing to assign labels! So the only super that should be labelling is the “active” one, which is the one that *most recently* issued the **activated** message. Also, only a supervisor that is active can authorize voters, or open and close the polls. These UI controls are hidden on all inactive supervisors.
 - State change: The Supervisor that sent this message becomes active and can authorize voters, and all other Supervisor machines become inactive and show an “Activate” button.
 - This message contains a list of the last-known status of every machine on the network that the supervisor knows about. If a machine was not in the list, or has a status update, it should in turn broadcast its own status.
 - The **status** message is a wrapper for a supervisor’s or votebox’s status, that contains the serial number of the machine that the status corresponds to. Since status messages are normally sent by themselves over Auditorium, they are implicitly tied to their sender. However, when contained within the activated message, the supervisor or votebox message has no notion of who sent it, thus the status wrapper. Status looks like: (**status** *node-id* (**supervisor**|**votebox**))
- (**polls-open?** *keyword*)
 - A query to ask other machines if they know whether the polls are open
 - Sent when a supervisor is activated, and doesn’t know if the polls are open
 - Upon receiving this message, a machine will check its logs to see if it thinks the polls are open, and then will reply with a **last-polls-open** message if so.

A.2.2 Sent by booths

- (**votebox** *label* *votebox-status* *battery-health* *protected-count* *public-count*)
 - Any inconsistencies are reported to the Supervisor (maybe).
 - *votebox-status* can be ready or in-use
 - *label* is the machine's label if it knows it already, otherwise it will report o for unlabeled
 - This message is sent once when the booth connects to Auditorium, and periodically as a status message.
 - This allows a booth to disconnect from the network, and upon reconnecting inform the supervisor(s) that it is still in use.

- (**last-polls-open** *polls-open-message*)
 - This is only necessary if we want other machines to inform the new active supervisor whether the polls are open.
 - The machines will check their own logs and report the last **polls-open** message they saw, iff it is not succeeded by a **polls-closed** message (in which case, the machine would simply not respond).
 - reply to **polls-open?**
 - If any machine responds, the supervisor will check that the message is a valid **polls-open** message, and that the *keyword* matches the one entered by the poll worker. If these conditions are met, the supervisor will silently change to polls opened status, and allow access to the authorize button.

- (**cast-ballot** *authorization-code* *encrypted-ballot*)
 - *Note*: Only used when the challenge scheme Chapter 4 is not in use. Otherwise, **commit-ballot** is used.
 - reply to **authorized-to-cast**
 - must happen between **polls-open** and **polls-closed**
 - State change: The Booth waits for a reply from the Supervisor that the ballot was received.

- (**commit-ballot** *authorization-code encrypted-ballot*)
 - *Note:* Only used when the challenge scheme Chapter 4 is in use. Otherwise, **cast-ballot** is used.
 - reply to **authorized-to-cast**
 - must happen between **polls-open** and **polls-closed**
 - State change: The Booth waits for a reply from the Supervisor that the ballot was received.

- (**cast-committed-ballot** *authorization-code*)
 - *Note:* Only used when the challenge scheme Chapter 4 is in use. Otherwise, **cast-ballot** is used.
 - This message is sent when the voter chooses *not* to challenge a vote.
 - must happen between **polls-open** and **polls-closed**
 - State change: The Booth waits for a reply from the Supervisor that the ballot was received.

- (**challenge** *authorization-code challenge response*)
 - *note:* only used when the challenge scheme Chapter 4 is in use. otherwise, **cast-ballot** is used.
 - This message is sent when the voter chooses to challenge a vote. The challenge response is a single integer (the one-time decryption key for the ballot corresponding to the *authorization-code*).
 - must happen between **polls-open** and **polls-closed**
 - state change: the booth waits for a reply from the supervisor that the challenge was received.

- (**override-cast-confirm** *authorization-code encrypted-ballot*)
 - reply to **override-cast**
 - must happen between **polls-open** and **polls-closed**
 - State change: The booth changes to “Ready” on the Supervisor, and the *authorization-code* is de-authorized. (If a **ballot-received** reply is expected, this is handled by that

message instead)

- (**override-cast-deny** *authorization-code*)
 - reply to **override-cast**
 - must happen between **polls-open** and **polls-closed**
 - State change: The Booth allows the voter to resume voting from where he left off.

- (**override-cancel-confirm** *authorization-code*)
 - reply to **override-cancel**
 - must happen between **polls-open** and **polls-closed**
 - State change: The booth changes to “Ready” on the Supervisor, and the *authorization-code* is de-authorized.

- (**override-cancel-deny** *authorization-code*)
 - reply to **override-cancel**
 - must happen between **polls-open** and **polls-closed**
 - State change: The Booth allows the voter to resume voting from where he left off.

APPENDIX B

HIGHLIGHT GRAPHICS

These graphics, summarizing the VOTEBOX project, were prepared in 2009 for use as “highlights” in NSF publications.

ACCURATE center researchers have participated in studies finding **serious flaws** in **current commercial DRE voting systems** that make them vulnerable to malfunctions or deliberate manipulation by attackers.

REPORT: UNSAFE!
comprehensive audits in California and Ohio

E-VOTE
Alice
Bob

malfunctions can destroy or reveal ballots

voting machine viruses possible

missing or incorrect use of cryptography

poor software engineering practices

electronic voting in peril

VoteBOX is an ACCURATE research project exploring **designs for new e-voting systems** that are trustworthy, reliable, and usable.

VoteBox records secure, **tamper-evident logs** with redundancy to survive failures and accidental deletion

**POWER ON
POLLS OPEN
BALLOT CAST
BALLOT CAST
BALLOT CAST**

CRYPTOGRAPHIC PROOFS ONE WAY

ALICE

DECRYPTING VERIFYING

RESULTS

anyone may **challenge** a VoteBox to produce verifiable proof of correct behavior (with the assistance of a third-party proof verifier of his choice)

auditors can use these logs to confirm the validity and integrity of ballots

voters enjoy a familiar and interactive e-voting user experience

on the web: accurate-voting.org & votebox.cs.rice.edu

APPENDIX C

INTERNET RESOURCES

More information about this project will be available for as long as possible at the following URLs:

- The VOTEBOX homepage, including source code, binary executables, and operating documentation for the VOTEBOX prototype, can be found at <http://votebox.cs.rice.edu/>.
- The VOTEBOX source code is hosted at <http://votebox.googlecode.com/>.
- This document, along with errata, will be archived by the author at <http://dsandler.org/research/thesis>.

BIBLIOGRAPHY

- [1] Tex. Elec. Code § 52.091, 1997.
- [2] Tex. Elec. Code § 52.094, 1997.
- [3] Help America Vote Act of 2002 (HAVA). Pub. L. 107-252, 2002.
- [4] Final Report of the Cuyahoga Election Review Panel, July 2006. http://cuyahogavoting.org/CERP_Final_Report_20060720.pdf.
- [5] B. Adida. Helios: Web-based open-audit voting. In *Proceedings of the 17th USENIX Security Symposium (Security '08)*, San Jose, CA, July 2008.
- [6] R. M. Alvarez, S. Ansolabehere, E. Antonsson, J. Bruck, S. Graves, T. Palfrey, R. Rivest, T. Selker, A. Slocum, and C. Stewart III. Voting: What Is, What Could Be. Report of the Caltech/MIT Voting Technology Project, July 2001.
- [7] R. M. Alvarez, M. Goodrich, T. E. Hall, D. R. Kiewiet, and S. M. Sled. The complexity of the California recall election. In *PS: Political Science and Politics*, volume 37, Nov. 2003.
- [8] R. M. Alvarez and T. E. Hall. *Electronic Elections: The Perils and Promises of Digital Democracy*. Princeton University Press, Princeton, New Jersey, 2008.
- [9] M. Bellare and B. Yee. Forward integrity for secure audit logs. Technical report, UC at San Diego, Dept. of Computer Science and Engineering, Nov. 1997.
- [10] J. Benaloh. Simple verifiable elections. In *Proceedings of the USENIX/ACCURATE Electronic Voting Technology Workshop (EVT '06)*, Vancouver, B.C., Canada, June 2006.
- [11] J. Benaloh. Ballot casting assurance via voter-initiated poll station auditing. In *Proceedings of the 2nd USENIX/ACCURATE Electronic Voting Technology Workshop (EVT '07)*, Boston, MA, Aug. 2007.
- [12] J. Bethencourt, D. Boneh, and B. Waters. Cryptographic methods for storing ballots on a voting machine. In *14th Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2007.
- [13] M. Blaze, A. Cordero, S. Engle, C. Karlof, N. Sastry, M. Sherr, T. Stegers, and K.-P. Yee. *Source Code Review of the Sequoia Voting System*. California Secretary of State's "Top to Bottom" Review, July 2007. http://www.sos.ca.gov/elections/voting_systems/ttbr/sequoia-source-public-jul26.pdf.
- [14] M. Blum, P. Feldman, and S. Micali. Non-interactive zero-knowledge and its applications. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 103–112, New York, NY, USA, 1988.
- [15] P. Boutin. Is e-voting safe? *PC World*, Apr. 28 2004. <http://www.pcworld.com/article/115608>.
- [16] K. W. Brace and M. P. McDonald. Final Report of the 2004 Election Day Survey. Submitted to the U.S. Election Assistance Commission, Sept. 2005. http://www.eac.gov/election_survey_2004/toc.htm.
- [17] J. L. Brunner. Report of findings. Project EVEREST: Evaluation and Validation of Election Related Equipment, Standards and Testing, Dec. 14th 2007. <http://www.sos.state.oh.us/sos/upload/everest/00-SecretarysEVERESTExecutiveReport.pdf>.
- [18] J. A. Calandrino, A. J. Feldman, J. A. Halderman, D. Wagner, H. Yu, and W. P. Zeller. *Source Code Review of the Diebold Voting System*. California Secretary of State's "Top to Bottom" Review, July 2007. http://www.sos.ca.gov/elections/voting_systems/ttbr/diebold-source-public-jul29.pdf.
- [19] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI'99)*, pages 173–186, New Orleans, LA, Feb. 1999.
- [20] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2), Feb. 1981.

- [21] D. Chaum. Secret-ballot receipts: True voter-verifiable elections. *IEEE Security & Privacy*, 2(1):38–47, Jan. 2004.
- [22] D. Chaum, R. Carback, J. Clark, A. Essex, S. Popoveniuc, R. L. Rivest, P. Y. A. Ryan, E. Shen, and A. T. Sherman. Scantegrity ii: End-to-end verifiability for optical scan election systems using invisible ink confirmation codes. In *Proceedings of the 3rd USENIX/ACCURATE Electronic Voting Technology Workshop (EVT '08)*, July 2008.
- [23] D. Chaum, A. Essex, R. Carback, J. Clark, S. Popoveniuc, A. Sherman, and P. Vora. Scantegrity: End-to-end voter-verifiable optical- scan voting. *IEEE Security & Privacy*, 6(3):40–46, May–June 2008.
- [24] D. Chaum and T. P. Pedersen. Wallet databases with observers. In E. F. Brickell, editor, *Advances in Cryptology – CRYPTO '92*, volume 740 of *Lecture Notes in Computer Science*, pages 89–105. Springer-Verlag, 1992.
- [25] D. Chaum, P. Y. A. Ryan, and S. A. Schneider. A practical, voter-verifiable election scheme. In *ESORICS '05*, pages 118–139, Milan, Italy, 2005.
- [26] M. Cherney. *Vote results further delayed*. The Sun News, Jan. 2008. <http://www.myrtlebeachonline.com/news/local/story/321972.html>.
- [27] M. R. Clarkson, S. Chong, and A. C. Myers. Civitas: A secure voting system. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.
- [28] Compuware Corporation, Columbus, OH. *Direct Recording Electronic (DRE) Technical Security Assessment Report*, Nov. 2003. <http://www.sos.state.oh.us/sos/hava/compuware112103.pdf>.
- [29] L. F. Cranor and R. K. Cytron. Sensus: A security-conscious electronic polling system for the internet. In *Proceedings of the Hawai'i International Conference on System Sciences*, Hawaii, USA, Jan. 1997.
- [30] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In *Advances in Cryptology – CRYPTO '89*, pages 307–315, Santa Barbara, CA, July 1989.
- [31] EAC Accredited Test Laboratories. United States Election Assistance Commission, Accessed April 2009. <http://www.eac.gov/program-areas/voting-systems/test-lab-accreditation/eac-accredited-test-laboratories/>.
- [32] Election Data Services. *2004 Voting Equipment Study*, Feb. 2004. http://www.electiondataservices.com/images/File/VotingEquipStudies%20%20ve2004_report.pdf (accessed April 2009).
- [33] S. Everett, K. Greene, M. Byrne, D. Wallach, K. Derr, D. Sandler, and T. Torous. Is newer always better? The usability of electronic voting machines versus traditional methods. In *Proceedings of CHI 2008*, Florence, Italy, Apr. 2008.
- [34] S. P. Everett. *The Usability of Electronic Voting Machines and How Votes Can Be Changed Without Detection*. PhD thesis, Rice University, Houston, TX, 2007.
- [35] A. J. Feldman, J. A. Halderman, and E. W. Felten. Security analysis of the Diebold AccuVote-TS voting machine. In *Proceedings of the 2nd USENIX/ACCURATE Electronic Voting Technology Workshop (EVT '07)*, Boston, MA, Aug. 2007.
- [36] E. Felten. *Diebold's Motherboard Flaw: Implications*. Princeton University, Feb. 2007. Freedom to Tinker (blog), <http://www.freedom-to-tinker.com/?p=1081#comment-179915>.
- [37] A. Fiat and A. Shamir. How to prove yourself: practical solutions to identification and signature problems. In *Advances in Cryptology – CRYPTO '86*, pages 186–194. Springer-Verlag, 1987.
- [38] K. Fisher, R. Carback, and T. Sherman. Punchscan: Introduction and system definition of a high-integrity election system. In *Workshop On Trustworthy Elections (WOTE 2006)*, Cambridge, U.K., June 2006.
- [39] A. Fujioka, T. Okamoto, and K. Ohta. A practical secret voting scheme for large scale elections. In *Advances in Cryptology – AUSCRYPT '92*, volume 718 of *Lecture Notes in Computer Science*, pages 244–251, Berlin, 1993. Springer-Verlag.
- [40] R. Gardner, S. Garera, and A. D. Rubin. Protecting against privacy compromise and ballot stuffing by eliminating non-determinism from end-to-end voting schemes. Technical Report 245631, Johns Hopkins University, Apr. 2008. http://www.cs.jhu.edu/~ryan/voting_randomness/ggr_voting_randomness.pdf.
- [41] S. N. Goggin and M. D. Byrne. An examination of the auditability of voter verified paper audit trail (VVPAT) ballots. In *Proceedings of the 2nd USENIX/ACCURATE Electronic Voting Technology Workshop (EVT '07)*, Berkeley, CA, USA, Aug. 2007.
- [42] K. K. Greene. Usability of an electronic voting interface: State information in a sequential navigation model. Presentation to HF/HCI Research Seminar (Psychology 531), Rice University, 2007.

- [43] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, Oct. 2007.
- [44] J. L. Hall and L. Quilter. Documentation review of the Hart InterCivic System 6.2.1 voting system, July 20 2007.
- [45] B. Harris. Inside a U.S. election vote counting program. Scoop, July 8th 2003. <http://www.scoop.co.nz/stories/HL0307/S00065.htm> (accessed April 2009).
- [46] B. Harris. Voting system integrity flaw. Scoop, Feb. 5th 2003. <http://www.scoop.co.nz/stories/HL0302/S00036.htm> (accessed April 2009).
- [47] M. A. Herschberg. Secure electronic voting using the World Wide Web. Master's thesis, Massachusetts Institute of Technology, June 1997.
- [48] S. Hertzberg. *DRE Analysis for May 2006 Primary, Cuyahoga County, Ohio*. Election Science Institute, San Francisco, CA, Aug. 2006. http://bocc.cuyahogacounty.us/GSC/pdf/esi_cuyahoga_final.pdf.
- [49] A. b. Huang. *Hacking the Xbox: An Introduction to Reverse Engineering*. No Starch Press, 2003.
- [50] H. Hursti. Critical security issues with Diebold optical scan design. Technical report, Black Box Voting, July 2005. <http://www.blackboxvoting.org/BBVreport.pdf>.
- [51] H. Hursti. Critical security issues with Diebold TSx. Technical report, Black Box Voting, May 2006. <http://www.blackboxvoting.org/BBVtsxstudy.pdf>.
- [52] InfoSENTRY Services. *Computerized Voting Systems Security Assessment: Summary of Findings and Recommendations*, Nov. 2003. <http://www.sos.state.oh.us/sos/hava/infoSentry112103.pdf>.
- [53] S. Inguva, E. Rescorla, H. Shacham, and D. S. Wallach. *Source Code Review of the Hart InterCivic Voting System*. California Secretary of State's "Top to Bottom" Review, July 2007. http://www.sos.ca.gov/elections/voting_systems/ttbr/Hart-source-public.pdf.
- [54] D. Jefferson, A. D. Rubin, and B. Simons. A comment on the May 2007 DoD report on voting technologies for UOCAVA citizens, June 2007. http://www.servesecurityreport.org/SERVE_Jr_v5.3.pdf.
- [55] D. Jefferson, A. D. Rubin, B. Simons, and D. A. Wagner. A security analysis of the secure electronic registration and voting experiment (SERVE), Jan. 2004. <http://www.servesecurityreport.org/>.
- [56] D. W. Jones. Problems with voting systems and the applicable standards. Testimony before the U.S. House of Representatives Committee on Science, May 2001. <http://www.cs.uiowa.edu/~jones/voting/congress.html>.
- [57] D. W. Jones. A brief illustrated history of voting, 2001 (updated 2003). <http://www.cs.uiowa.edu/~jones/voting/pictures/>.
- [58] D. W. Jones. The evaluation of voting technology. In D. Gritzalis, editor, *Secure Electronic Voting (Advances in Information Security)*. Springer-Verlag, 2002.
- [59] D. W. Jones. The Case of the Diebold FTP site, 2003. <http://www.cs.uiowa.edu/~jones/voting/dieboldftp.html> (accessed April 2009).
- [60] D. W. Jones. Parallel testing during an election, 2004. <http://www.cs.uiowa.edu/~jones/voting/testing.shtml#parallel>.
- [61] D. W. Jones and T. C. Bowersox. Secure data export and auditing using data diodes. In *Proceedings of the USENIX/ACCURATE Electronic Voting Technology Workshop (EVT '06)*, Vancouver, B.C., Canada, Aug. 2006.
- [62] C. Karlof, N. Sastry, and D. Wagner. Cryptographic voting protocols: A systems perspective. In *USENIX Security Symposium*, Aug. 2005.
- [63] J. Kelsey, A. Regenscheid, T. Moran, and D. Chaum. Hacking paper: Some random attacks on paper-based E2E systems. Presentation in Seminar 07311: Frontiers of Electronic Voting, 29.07.07–03.08.07, organized in The International Conference and Research Center for Computer Science (IBFI, Schloss Dagstuhl, Germany), Aug. 2007. <http://kathrin.dagstuhl.de/files/Materials/07/07311/07311.KelseyJohn.Slides.pdf>.
- [64] A. Kiayias, M. Korman, and D. Walluck. An Internet voting system supporting user privacy. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 165–174, Washington, DC, USA, 2006.
- [65] A. Kiayias and M. Yung. The vector-ballot e-voting approach. In *FC'04: Financial Cryptography 2004*, Key West, FL, Feb. 2004.

- [66] T. Kohno, A. Stubblefield, A. D. Rubin, and D. S. Wallach. Analysis of an electronic voting system. In *Proc. of IEEE Symposium on Security & Privacy*, Oakland, CA, 2004.
- [67] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.
- [68] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions Programming Languages and Systems*, 4(3):382–401, 1982.
- [69] P. Maniatis and M. Baker. Secure history preservation through timeline entanglement. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, Aug. 2002.
- [70] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, Apr. 1960.
- [71] D. Molnar, T. Kohno, N. Sastry, and D. Wagner. Tamper-evident, history-independent, subliminal-free data structures on PROM storage -or- How to store ballots on a voting machine (extended abstract). In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2006.
- [72] T. Moran and M. Naor. Receipt-free universally-verifiable voting with everlasting privacy. In *CRYPTO '06: Proceedings of the 26th International Cryptology Conference*, Santa Barbara, CA, Aug. 2006.
- [73] M. Naor and A. Shamir. Visual cryptography. In A. De Santis, editor, *Advances in Cryptology – EUROCRYPT '94*, volume 950 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 1995.
- [74] C. A. Neff. A verifiable secret shuffle and its application to e-voting. In *CCS '01: Proceedings of the 8th ACM Conference on Computer and Communications Security*, pages 116–125, Philadelphia, PA, 2001.
- [75] Project EVEREST (Evaluation and Validation of Election-Related Equipment, Standards, and Testing). *Voting System Review Findings*, Dec. 2007. <http://www.sos.state.oh.us/SOS/elections/voterInformation/equipment/VotingSystemReviewFindings.aspx> (accessed April 2009).
- [76] RABA Technologies, Columbia, MD. *Trusted Agent Report: Diebold AccuVote-TS Voting System*, Jan. 2004. http://www.raba.com/press/TA_Report_AccuVote.pdf.
- [77] R. L. Rivest. S-expressions. IETF Internet Draft, May 1997. <http://people.csail.mit.edu/rivest/sexp.txt>.
- [78] R. L. Rivest. The ThreeBallot voting system. <http://theory.csail.mit.edu/~rivest/Rivest-TheThreeBallotVotingSystem.pdf>, Oct. 2006.
- [79] R. L. Rivest and W. D. Smith. Three voting protocols: ThreeBallot, VAV, and Twin. In *Proceedings of the 2nd USENIX/ACCURATE Electronic Voting Technology Workshop (EVT'07)*, Boston, MA, Aug. 2007.
- [80] R. L. Rivest and J. P. Wack. On the notion of “software independence” in voting systems, 2006. <http://vote.nist.gov/SI-in-voting.pdf>.
- [81] D. Rohde. On New Voting Machine, the Same Old Fraud. *The New York Times*, April 27, 2004. <http://www.nytimes.com/2004/04/27/international/asia/27indi.html>.
- [82] P. Y. A. Ryan and T. Peacock. A threat analysis of Prêt à Voter. In *Workshop On Trustworthy Elections (WOTE 2006)*, Cambridge, U.K., June 2006.
- [83] K. Sako and J. Kilian. Receipt-free mix-type voting scheme - a practical solution to the implementation of a voting booth. In *Advances in Cryptology – EUROCRYPT '95*, volume 921 of *Lecture Notes in Computer Science*, pages 393–403. Springer-Verlag, 1995.
- [84] D. Sandler, K. Derr, S. Crosby, and D. S. Wallach. Finding the evidence in tamper-evident logs. In *Proceedings of the 3rd International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE'08)*, Oakland, CA, May 2008.
- [85] D. R. Sandler. Feedtree: Scalable and prompt delivery for web feeds. Master’s thesis, Rice University, Houston, TX, Apr. 2007.
- [86] D. R. Sandler, K. Derr, and D. S. Wallach. VoteBox: A tamper-evident, verifiable electronic voting system. In *Proceedings of the 17th USENIX Security Symposium (Security '08)*, San Jose, CA, July 2008.
- [87] D. R. Sandler and D. S. Wallach. Casting votes in the Auditorium. In *Proceedings of the 2nd USENIX/ACCURATE Electronic Voting Technology Workshop (EVT '07)*, Boston, MA, Aug. 2007.
- [88] D. R. Sandler and D. S. Wallach. The case for networked remote voting precincts. In *Proceedings of the 3rd USENIX/ACCURATE Electronic Voting Technology Workshop (EVT '08)*, San Jose, CA, Aug. 2008.

- [89] D. R. Sandler and D. S. Wallach. Birds of a feather: Open, decentralized micropublishing. In *Proceedings of the 8th International Workshop on Peer-to-Peer Systems (IPTPS '09)*, Boston, MA, Apr. 2009. *To appear*.
- [90] N. Sastry, T. Kohno, and D. Wagner. Designing voting machines for verification. In *Proceedings of the 15th USENIX Security Symposium*, Vancouver, B.C., Canada, Aug. 2006.
- [91] B. Schneier and J. Kelsey. Cryptographic support for secure logs on untrusted machines. In *USENIX Security Symposium*, pages 53–62, San Antonio, TX, Jan. 1998.
- [92] B. Schneier and J. Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information and System Security*, 1(3), 1999.
- [93] Science Applications International Corporation (SAIC), Annapolis, MD. *Risk Assessment Report: Diebold AccuVote-TS Voting System and Processes*, Sept. 2003. Redacted form: <http://www.verifiedvoting.org/downloads/votingsystemreportfinal.pdf>, partially unredacted and discussed at <http://www.bradblog.com/?p=3719>.
- [94] Securities and Exchange Commission, Washington, D.C. *SEC charges four brokers and day trader in fraudulent "squawk box" scheme*, Aug. 2005. <http://www.sec.gov/news/press/2005-114.htm>.
- [95] A. Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.
- [96] R. M. Stein and G. Vonnahme. Election day vote centers and voter turnout. Prepared for presentation at the 2006 Annual Meetings of the Midwest Political Science Association, Apr. 2006.
- [97] TheSpecialist. 360 FW hacked. XboxHacker BBS, Mar. 2006. http://www.xboxhacker.net/index.php?option=com_smf&Itemid=33&topic=481.0.
- [98] A. H. Trechsel, G. Schwerdt, F. Breuer, R. M. Alvarez, and T. E. Hall. Internet voting in the March 2007 parliamentary elections in Estonia, July 2007. <http://www.vote.caltech.edu/reports/EvotingEstonia2007.pdf>.
- [99] United States Election Assistance Commission. *Voluntary Voting System Guidelines*, 2005.
- [100] H. A. von Spakovsky and R. Buhler. Disenfranchised over there. *The Weekly Standard*, 013(33), May 2008.
- [101] D. S. Wallach. Security and Reliability of Webb County's ES&S Voting System and the March '06 Primary Election. Expert Report in *Flores v. Lopez*, <http://accurate-voting.org/wp-content/uploads/2006/09/webb-report2.pdf>, May 2006.
- [102] S. Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.
- [103] A. Yasinsac, D. Wagner, M. Bishop, T. Baker, B. de Medeiros, G. Tyson, M. Shamos, and M. Burmester. *Software Review and Security Analysis of the ES&S iVotronic 8.0.1.2 Voting Machine Firmware*. Security and Assurance in Information Technology Laboratory, Florida State University, Tallahassee, Florida, Feb. 2007. <http://election.dos.state.fl.us/pdf/FinalAudRepSAIT.pdf>.
- [104] K.-P. Yee. Extending prerendered-interface voting software to support accessibility and other ballot features. In *Proceedings of the 2nd USENIX/ACCURATE Electronic Voting Technology Workshop (EVT '07)*, Boston, MA, Aug. 2007.
- [105] K.-P. Yee, D. Wagner, M. Hearst, and S. M. Bellovin. Prerendered user interfaces for higher-assurance electronic voting. In *Proceedings of the USENIX/ACCURATE Electronic Voting Technology Workshop (EVT '06)*, Vancouver, B.C., Canada, 2006.
- [106] A. R. Yumerefendi and J. S. Chase. Strong accountability for network storage. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '07)*, pages 77–92, San Jose, CA, Feb. 2007.

COLOPHON

This document was composed in `vim` (specifically `MacVim.app`¹ and typeset using $X_{\text{Y}}\text{T}_{\text{E}}\text{X}$ ², Jonathan Kew's Unicode- and OpenType-aware variant of $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$. The `BibTEX`-formatted bibliography was managed with the help of the open-source `BibDesk`³ application.

Body type is set in `Minion` (Robert Slimbach, 1990); headings, runners, and other sans-serif elements are set in `Avenir` (Adrian Frutiger, 1998); monospaced type and code are set in `Vera Sans Mono` (Jim Lyles, 2003). The `VoteBox` logo (see next page) is set in `ITC Avant Garde Gothic` (Herb Lubalin, ca. 1971).

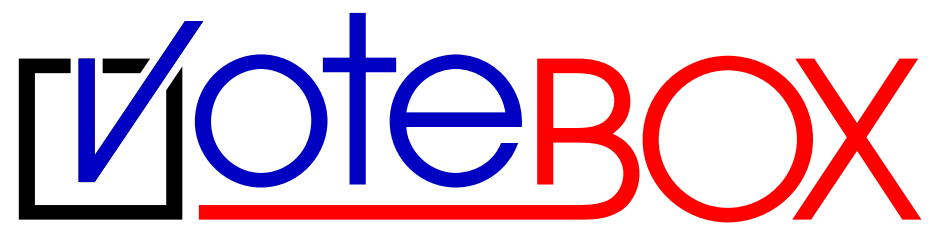
Vector graphics were created in `Adobe Illustrator`; bitmap artwork was tweaked in `Gus Mueller's Acorn`⁴ image editor. All this was made on a Mac.

¹<http://macvim.googlecode.com/>

²<http://www.tug.org/xetex/>

³<http://bibdesk.sourceforge.net/>

⁴<http://flyingmeat.com/acorn/>

The logo for VoteBOX features a black square on the left containing a blue checkmark. To the right of the square, the word "Vote" is written in blue lowercase letters, and "BOX" is written in red uppercase letters. A red horizontal line underlines the "Vote" portion of the text.