

Building Reliable Voting Machine Software

Ka-Ping Yee



Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2007-167

<http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-167.html>

December 19, 2007

Copyright © 2007, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

I am grateful to many people who helped make this dissertation possible.

My advisors: David Wagner, Marti Hearst.

My committee members: Henry Brady, Joe Hellerstein.

Advice: Steve Bellovin, Candy Lopez, Scott Luebking, Noel Runyan, Joseph Hall.

Security review: Matt Bishop, Ian Goldberg, Tadayoshi Kohno, Mark Miller, Dan Sandler, Dan Wallach.

Funding: National Science Foundation, through ACCURATE.

Thanks also to Scott Kim, La Shana Porlaris, Lisa Friedman, and my parents.

Building Reliable Voting Machine Software

Ka-Ping Yee

B. A. Sc. (University of Waterloo) 1998

A dissertation submitted to the Graduate Division
of the University of California, Berkeley
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in
Computer Science

Committee in charge:

Professor David Wagner, Co-chair

Professor Marti Hearst, Co-chair

Professor Henry Brady

Professor Joseph Hellerstein

Fall 2007

The dissertation of Ka-Ping Yee is approved.

Professor David Wagner (Co-chair)

Date

Professor Marti Hearst (Co-chair)

Date

Professor Henry Brady

Date

Professor Joseph Hellerstein

Date

University of California, Berkeley
Fall 2007

Building Reliable Voting Machine Software

Copyright © 2007

Ka-Ping Yee

Permission is granted to copy, distribute, and/or modify this document under the terms of the GNU Free Documentation License, version 1.2 or any later version published by the Free Software Foundation, with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the appendix entitled *GNU Free Documentation License*.

Abstract

Building Reliable Voting Machine Software

Ka-Ping Yee

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David Wagner, Co-chair

Professor Marti Hearst, Co-chair

I examine the question of how to design election-related software, with particular attention to the threat of insider attacks, and propose the goal of simplifying the software in electronic voting machines. I apply a technique called *prerendering* to reduce the security-critical, voting-specific software by a factor of 10 to 100 while supporting similar or better usability and accessibility, compared to today's voting machines. Smaller and simpler software generally contributes to easier verification and higher confidence.

I demonstrate and validate the prerendering approach by presenting Pvote, a vote-entry program that allows a high degree of freedom in the design of the user interface and supports synchronized audio and video, touchscreen input, and input devices for people with disabilities. Despite all its capabilities, Pvote is just 460 lines of Python code; thus, it directly addresses the conflict between flexibility and reliability that underlies much of the current controversy over electronic voting. A security review of Pvote found no bugs in the Pvote code and yielded lessons on the practice of adversarial code review. The analysis and design methods I used, including the prerendering technique, are also applicable to other high-assurance software.

Professor David Wagner

Professor Marti Hearst

This dissertation is dedicated to those who work to run elections everywhere in the world: registrars, officers, pollworkers, clerks, judges, scrutineers, observers, and everyone else involved in the process. You carry out the mechanisms that make democracy work; this research is devoted to helping you make democracy work better.

Preface

The democracy upon which our modern society is built ultimately depends on a system that collects and counts votes. For many voters in the United States and other countries, nearly every part of that system relies on computer software in some way. If you had to design that software, how would you do it?

This dissertation offers an exploration of that question and a proposed answer: create the simplest possible voting machine software. I use a technique called *prerendering* to **reduce the critical voting-specific software by a factor of 10 to 100 while supporting similar or better accessibility and usability**, compared to today's machines. Central to this dissertation is the story of Pvote, the program I developed to realize this goal.

The first reason to simplify software is the threat of an *insider attack*. The challenge is to prevent not just inadvertent flaws, but flaws *intentionally* crafted by programmers who stand to gain from subverting their own software. The only way to meet this challenge is to require simpler software.

The second reason is that much of the controversy over electronic voting stems from a conflict between flexibility and reliability. Computers offer the promise of broader and more effective access to voting, but computer programs are more complicated and fragile than hand-counted paper ballots. Simplifying the voting machine software mitigates this dilemma.

The problem of electronic voting is illustrative of the challenges of building reliable software in general. In particular, I report on insights from the Pvote work about managing the complexity of high-assurance software and about reviewing software for correctness without assuming trust in its author. Both are relevant to the prevention of insider attacks, which are a thorny and long-standing problem in software security.

This dissertation is intended for several audiences:

- *Election staff, policymakers, and activists*: If you run elections or influence how elections are conducted, I hope to make you aware of the perils of complexity in software (Chapters 1 and 9), and to calibrate your tolerance for complexity in election software by demonstrating how much it can be simplified. I also hope to contribute to your understanding of the tradeoffs among various choices of voting equipment and verification methods (Chapter 3).
- *Engineers*: If you build software, you may be able to achieve greater confidence in it using the analysis, design, and review strategies presented here (Chapters 2, 3, and 8). If you develop voting machines, you can apply the prerendering strategy to create more reliable software (Chapter 4), use ideas from Pvote's design and implementation (Chapters 5, 6, and 7), or use the Pvote code as a basis for your own software (Appendices A and B).
- *Researchers*: If you investigate software reliability or security, you may be interested in *assurance trees* (Chapter 2), a way of structuring assurance claims during software design, *prerendering* (Chapter 4) as a strategy for reducing the trusted code base of a system, or *derivation maps* (Chapter 9) for understanding sources of vulnerability to insiders and the effects of shifting complexity among components. The Pvote review experience (Chapter 8 and Appendix E) motivates research challenges in the design of programming languages, development environments, and reviewing tools to support adversarial code review.
- *Designers*: If you practice visual design or interaction design, you may be interested to learn how *prerendering* (Chapter 4), the main software approach presented here, can offer you unprecedented freedom in designing electronic ballots and new opportunities for advancing democracy through the power of design.

Contributions

This is a quick guide to the main contributions of this work and where to find them.

1. A set of *correctness properties for voting software* derived as an assurance tree (page 24).
2. An *assurance chart comparing types of voting systems* according to the verification mechanisms available to voters at each step of the voting process (page 56).
3. *User interface prerendering*, a technique for reducing the complexity of critical software components (page 57).
4. Pvote's *ballot definition file format*, a platform-independent format for describing the ballot and the voting user interface in a prerendered user interface voting system (page 121).
5. The software design of *Pvote*, a vote-entry program with support for a wide range of ballot designs and voters with disabilities (page 127).
6. A set of *desirable properties of programming languages* to support adversarial code review (page 149).
7. *Lessons learned from the Pvote security review*, the first open adversarial code review of voting software designed for minimal complexity and high assurance (page 153).
8. *Derivation mapping*, a method of diagramming the provenance of a security-critical artifact to identify sources of vulnerability to insider attacks (page 161).
9. A *security argument for the use of high-level programming languages* in high-assurance software (page 173).
10. *Proof by construction* (the implementation of Pvote) that a fully featured user interface for voting can be implemented in 460 lines of Python (page 217).
11. A *security analysis* and a set of assurance arguments for Pvote, which are given in a separate document [92].

Acknowledgements

I have been extremely lucky to have David Wagner and Marti Hearst as my advisors. They supervised and supported this work, and provided me with guidance and insight during my career as a graduate student. They removed obstacles and sought out opportunities for me. Their responsiveness and detailed feedback have been fantastic. I also thank Henry Brady and Joe Hellerstein, who served on my committee and went out of their way to review this dissertation on a short time frame.

Steve Bellovin suggested the idea of prerendering, which sparked this work. Candy Lopez of the Contra Costa County Elections Department patiently showed me how real elections are run. Scott Luebking and Noel Runyan helped me understand the accessibility issues surrounding voting. Matt Bishop, Ian Goldberg, Tadayoshi Kohno, Mark Miller, Dan Sandler, and Dan Wallach generously volunteered many, many hours of their time to serve as expert reviewers in the Pvote security review. Joseph Hall has been a wonderful resource on election policy.

Debra Bowen and David Wagner created and gave me the rare opportunity to review the source code of a widely used commercial voting system in the California Top-to-Bottom Review. It was a privilege to work with my collaborators on that project: Matt Blaze, Arel Cordero, Sophie Engle, Chris Karlof, Naveen Sastry, Micah Sherr, and Till Stegers.

Public attention to electronic voting did not appear overnight; it is the result of a long history of hard work by civic-minded heroes such as David Dill (founder of the Verified Voting Foundation), Avi Rubin (director of ACCURATE), and many others. Their efforts are a big part of what has made research like mine possible. This work was funded by the National Science Foundation, through ACCURATE.

Mark Miller, Jonathan Shapiro, and Marc Stiegler sparked my interest in computer security and have deeply shaped my understanding of it through many years of fruitful collaboration and shared wisdom. I am exceptionally fortunate to have met and worked with them.

Scott Kim's dissertation inspired the page design of this dissertation. La Shana Porlaris of the EECS Department saved me from crisis time and again; her help and calm advice were invaluable.

I am especially grateful to Lisa Friedman for her support during the writing of this dissertation, and to my parents, for a lifetime of devotion to me and my education.

Contents

Preface	ii
Contributions	iv
Acknowledgements	v
Contents	x
1 Voting	1
What makes the voting problem so hard?	2
How does an election work?	6
Why use computers for elections?	9
How did electronic voting become controversial?	11
Why does software correctness matter?	14
2 Correctness	16
What constitutes a democratic election?	17
What does it mean for a voting system to be correct?	19
How does correctness relate to safety?	20
What is the tree of assurance goals for an election?	24
What does it mean for a voting system to be secure?	30
3 Verification	33
How do we gain confidence in election results?	34
How can we verify the computerized parts of an election?	36
What kind of election data can be published?	39
What makes software hard to verify?	41
In what ways are today's voting systems verifiable?	44
What is the minimum software that needs to be verified?	48
What other alternatives for verification are possible?	52

4 Prerendering	57
How can we make vote-entry software easier to verify?	58
What is prerendering?	59
Why put the entire user interface in the ballot definition? . .	60
How would a voting computer use a prerendered ballot? . .	62
What is gained by publishing the ballot definition?	63
What are the advantages of prerendering?	65
How can prerendering be applied to other software?	66
How are votes recorded anonymously?	67
5 Ptouch: the touchscreen prototype	69
Overview	70
Ballot definition format	71
Software design	80
Implementation	83
Evaluation	88
Shortcomings	93
6 Accessibility	96
Why was a second prototype needed?	97
What is Pvote’s approach to accessibility?	98
How are alternative input devices handled?	99
How does blindness affect interface navigation?	100
How do blind users stay oriented within an interface?	101
How do blind users keep track of what is selected?	102
How do blind users get feedback on their actions?	103
How are vision-impaired users accommodated?	104
7 Pvote: the multimodal prototype	105
Overview	106
Goals	107
Design principles	110
Differences between Pvote and Ptouch	114
Ballot definition format	121
Software design	127
Implementation	132
Evaluation	133

8	Security review	136
	How was Pvote’s security evaluated?	137
	What were Pvote’s security claims?	139
	How was Pthin defined?	143
	What flaws did the reviewers find?	145
	What improvements did the reviewers suggest?	146
	Did the reviewers find the inserted bugs?	148
	What ideas did reviewers have on programming languages?	149
	What ideas did reviewers have on conducting reviews?	151
	What lessons were learned from the review?	153
9	Complexity	156
	Does prerendering actually eliminate complexity?	157
	What is achieved by shifting complexity?	158
	Why do software reviews assume trust in compilers?	160
	How far back can the derivation of a program be traced?	161
	What affects the tolerance of complexity in a component?	164
	How does Pvote reallocate complexity?	167
	What is gained by using interpreted languages?	173
10	Related work	174
	Do any other voting systems use prerendering?	175
	What other voting proposals reduce reliance on software?	176
	What are “frog” voting systems?	177
	Do frogs solve the electronic voting problem?	178
	What is “software independence” (SI)?	179
	Does SI make software reliability irrelevant?	181
	What is end-to-end (E2E) verification?	186
	Does E2E verification make software reliability irrelevant?	187
	What are other approaches to high-assurance software?	188
	Conclusion	191
	Bibliography	193
A	Ptouch source code	204
	main.py	205

Ballot.py	206
Navigator.py	210
Video.py	214
Recorder.py	215
B Pvote source code	217
main.py	218
Ballot.py	220
verifier.py	224
Navigator.py	228
Audio.py	233
Video.py	235
Printer.py	236
C Sample Pvote ballot definition	237
D Sample Pvote ballot designs	267
E Pvote security review findings	272
Correctness	273
Consensus recommendations	278
Inconclusive recommendations	282
Observations	284
Open issues	288
Bug insertion	296
Review process	300
Post-review survey	304
GNU Free Documentation License	306

1 Voting

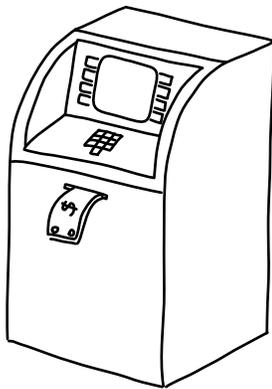
What makes the voting problem so hard?	2
How does an election work?	6
Why use computers for elections?	9
How did electronic voting become controversial?	11
Why does software correctness matter?	14

What makes the voting problem so hard?

When I say the “voting problem,” I’m referring specifically to the system that collects and counts votes. There are many other parts of the election process that I’m not going to address in this dissertation, such as voter registration, electoral systems, and election campaigning. The collection and counting of votes has been particularly controversial in the United States due to problems with electronic voting in recent elections.

One of the great things about doing election-related research is that just about everyone immediately understands why it’s important. In my experience, whenever elections are the topic of conversation, people have a lot to say about their opinions on the matter. It’s encouraging to see that so many people care deeply about democracy.

In conversations about the voting problem, there seem to be four ideas in particular that come up all the time. It’s not unusual to think that running a fair election ought to be a straightforward task—after all, in some sense, it’s just counting. To give you a taste of why the voting problem is not as easy as it might seem, let’s begin by examining these four suggestions.

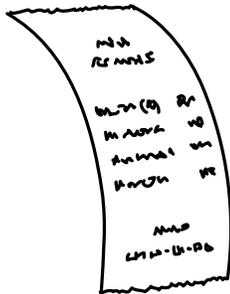


Banking machines work fine, so voting machines should be no problem. On the surface, banking machines and voting machines seem similar: users walk up and make selections on a touchscreen to carry out a transaction. One of the largest vendors, Diebold Inc., even produces both kinds of machines. But the incentives and risks are very different.

Banking machines have money inside—the bank’s money. If money goes missing, you can bet the bank will find out right away and be strongly motivated to fix the problem. If the bank machine incorrectly gives out too much cash, the bank loses money; if it gives out too little, the bank will be dealing with irate customers. Everything about the bank transaction is recorded, from the entries in your bank statement to the video recorded by the camera in most bank machines. That’s because

the bank has a strong incentive to audit that money and track where it goes. If the machine makes mistakes, the bank loses—either they expend time and money correcting your problem, or you will probably leave and take your business to another bank.

With voting machines, it's another story altogether. Voting machines aren't supposed to record video or keep any record that associates you with your votes, because your ballot is supposed to be secret. You don't receive any tangible confirmation that your vote was counted, so you can't find out if there's a problem. Anybody can stand to gain by causing votes to be miscounted—a voter, pollworker, election administrator, or voting machine programmer—and the consequences are much harder to reverse. Correcting an error in your bank balance is straightforward, but the only way to fix an improperly counted election is to do an expensive manual recount or run the whole election again. And if you're unhappy with the way your vote was handled, you can't easily choose to vote on a competitor's machine.



Give each voter a printed receipt, just like we do for any other transaction. The surface comparison between voting and a financial transaction also leads many people to suggest that receipts are the answer. But the purpose of a receipt is quite different from what is needed to ensure an accurate election.

When you buy something, the receipt confirms that you paid for it. If there turns out to be a problem with the product, you can use the receipt to get your money back or to get the defective product exchanged.

When talking about a receipt from a voting machine, what most people have in mind is a printed record of the choices you made, just like a receipt from a cash register. If you took home such a receipt, what would you do with it? There's nothing to refund, and you can't use a receipt to get an exchange on a defective politician. The receipt could record the choices you made, but the receipt alone doesn't assure that those choices were counted in the final result. In fact, if the receipt constitutes proof of which choices you made, it can be sold—

defeating the whole point of the secret ballot, which is to avoid the corruption that vote-buying campaigns can cause.

A truly useful voting “receipt” would do exactly the opposite: it would *not* reveal which choices you made but *would* let you confirm that your choices were counted. Although these two requirements sound paradoxical, researchers have invented a variety of schemes that achieve them through the clever use of cryptography. However, a key weakness of the schemes proposed so far is that they rely on advanced mathematics, with a counting process that would be a mystery to all but a tiny minority of voters. This would run counter to the democratic principle of transparent elections. Researchers are continuing to search for simpler verification schemes that can be understood by an acceptably large fraction of the public.

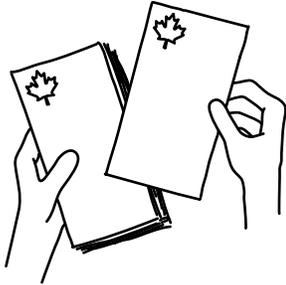


If we can trust computers to fly airplanes, we can trust computers to run elections. The comparison between airplanes and elections misses at least three key differences.

First, the visibility of failure is different. An airplane cannot secretly fail to fly. When an airplane crashes, it makes headlines; everybody knows. A forensic investigation takes place, and if the crash is due to a manufacturing defect, the airplane manufacturer may be sued for millions of dollars. But an election system can produce incorrect results without any obvious signs of failure. Therefore, we require something more from election system software than what we require from airplane software. A successful election system must not only *work* correctly; it must also allow the public to *verify* that it worked correctly.

Second, the target audience is different. Commercial airplanes are designed to be flown by pilots with expert training, but voting machines have to be set up by pollworkers and operated by the general public. Our trust in airplanes is a combination of trust in the equipment and trust in the pilots who operate it. Whereas pilots have to log hundreds of hours of flight time to get a license, pollworkers are often hired on a temporary basis with only an afternoon or a day of training.

Third, security violations affect the perpetrators differently. Pilots and flight attendants are strongly motivated to uphold security procedures because their own lives could be at risk. A rogue voter or pollworker, on the other hand, would have more to gain and less to lose by surreptitiously changing the outcome of an election.



Count the ballots by hand—it works for the Canadians.

Ballots are considerably longer and more complicated in the United States than in many other countries. Whereas there is just one contest in a Canadian federal election (each voter selects a Member of Parliament), ballots in the United States can contain dozens of contests. For example, a typical ballot¹ for the November 2004 general election in Orange County, California contained 7 offices and 16 referenda, for a total of 23 contests that would have to be tallied by hand. Ballots in Chicago, Illinois that year² were even longer: ten pages of selections, consisting of 15 elected offices, confirmations of 74 sitting judges, and one referendum—a total of 90 contests. When you appreciate the scale of the task, it becomes easier to understand why many people are motivated to automate the process with computers. Hand-counting paper ballots is by no means impossible, but it would be considerably more expensive and time-consuming in the United States than in other countries with simpler ballots.

* * *

In summary, voting is especially challenging because:

- All involved parties can gain by corrupting an election.
- Results can be incorrect without an obvious failure.
- Democracy demands verifiability, not just correctness.
- Voter privacy and election transparency are in conflict.
- Elections must be accessible and usable by the public.
- Ballots in the United States are long and complex.

¹The example here is Orange County’s ballot type SB019 from November 2004, available in NIST’s collection of sample ballots at <http://vote.nist.gov/ballots.htm>.

² This refers to the “Code 9” ballot style in Cook County, Illinois (also available in NIST’s collection), used in Ward 19, Precincts 28, 43(R), 48, 50(R), and 66, as well as precincts in Wards 21 and 34.

How does an election work?

Running an election is a tremendous organizational task. In the end, it does come down to counting, but it's what's being counted that makes it such a challenge. Election administrators are, in effect, trying to take a fair and accurate measurement of the preferences of the entire population—a controlled experiment on a grand scale. As any psychologist will tell you, performing experimental measurements on human subjects is fraught with logistic pitfalls and sources of error. But elections are worse: virtually everybody has an incentive to actively bias the measurement toward their own preferred outcome. Thus, elections involve a security element as well, unlike most scientific measurements.

As if that weren't enough, a typical election in the United States is not just one opinion poll but many different polls conducted on the same day—for federal, state, and local elected offices, as well as state and local referenda—and each poll has to be localized to a specific region. Each contest appears on some ballots but not others, resulting in different combinations of contests on different ballots. Each combination is called a *ballot style*. Because there are so many kinds of districts (such as congressional districts, state assembly districts, municipalities, hospital districts, and school districts), and district boundaries of each kind often run through districts of other kinds, there can be over a hundred different ballot styles in a single county. There can also be multiple ballot styles at one polling place, if it serves voters on both sides of a district boundary, or if there are different ballots for voters of different political parties.

Process. Here is a simplified breakdown of the election process, setting aside voter registration and considering only the collection and counting of votes. The events before, during, and after actual voting make up the three stages of the process: *preparation, polling, and counting*.

- *Preparation.* Before any votes can be cast, election officials must prepare the ballots. Election officials map out all the different kinds of political districts, assemble the contests that are relevant to each political district, compose the contests into ballot styles, and determine which ballot styles go to which polling places.
- *Polling.* At polling places, pollworkers sign in each voter and make sure that each voter gets the correct style of ballot. Each voter makes their selections privately and casts a ballot. Voters may also have the option of voting by mail or participating in “early voting” by showing up in person at a special polling place before election day.
- *Counting.* The records of cast votes are counted, either at the polling places or at a central election office. If counting initially occurs at polling places, the counts are then transmitted to the central office for tallying. The votes for each contest are extracted from all the ballots on which that contest appears, and tallied to produce a result.

Equipment. The preceding description is intentionally ambiguous about whether paper or electronic voting is used, because the same three stages take place regardless of the type of equipment.

If paper ballots are used, a layout is prepared for each ballot style, usually designed on a computer. Election administrators estimate how many ballots of each style will be needed so that an adequate number can be printed for distribution to polling places. After being marked, paper ballots can be counted by hand or scanned on machines (called *optical scanning* machines). The scanning can take place at the polls (*precinct count optical scanning*), where each voter feeds their ballot through a scanning machine into a ballot box, or it can take place at a central office, where all the paper ballots are gathered and scanned in high-speed machines after polls close (*central count optical scanning*).

An alternative to paper ballots is to make selections on an electronic voting machine that directly records the selections in

computer memory. These machines are called *direct recording electronic* (DRE) machines. In this case, preparing ballots consists of producing *ballot definition* files on electronic media (such as memory cards or cartridges) to be placed in voting machines. The ballot definition determines what will be displayed to the voter. (Machines for scanning paper ballots also require ballot definitions that specify how the marks on the paper should be counted.) Some DRE machines also print a *voter-verified paper audit trail* (VVPAT)—a paper record of the voter's selections that is shown to the voter for confirmation, but kept sealed inside the machine to enable later recounts.

* * *

To sum up, there are three broad categories of elections in terms of how machines are used:

1. Vote on paper; count by hand.
 2. Vote on paper; count by machine.
 3. Vote on machine; count by machine.
- (3a. The voting machine may also produce a paper record.)

Why use computers for elections?

As the preceding description makes clear, all three stages of the election process involve complex and detail-oriented work. Preparation involves managing information about all the different contests, candidates, and ballot styles. Polling involves distributing this information and collecting results from all the polling places. Counting involves consolidating all the votes for each candidate in each contest across all the ballots and ballot styles. With so many contests on the ballot, computers can make this process much easier.

It's not surprising that election administrators have looked to computers for help with elections. Computers are used to great benefit in automating a broad range of complex and repetitive tasks and for recordkeeping functions throughout all kinds of government agencies. Running an election involves organizing and processing a lot of information, such as ballot descriptions and vote tallies, and databases are effective tools for managing this information.

The appeal of computers goes beyond their potential to increase the speed and accuracy of the count. Computerized vote-entry machines have much greater flexibility than paper ballots in the method of presenting contests and choices to voters. They can walk voters through the voting process, provide more detailed instructions, and prevent overvotes. They eliminate the possibility of ambiguous or improperly scanned marks on paper. They can offer a larger selection of languages. They can point out contests that a voter may have missed before finalizing the marked ballot. They can even read the names of candidates aloud, in headphones, for voters who have trouble reading or voters who are blind. Some voters have physical disabilities that prevent them from using pencil and paper. Computerized vote-entry machines allow people to vote using a variety of input devices, such as large buttons, foot pedals, head-controlled switches, or switches controlled by air pressure ("sip-and-puff" devices).

All of these things become possible when the voting process is conducted by an interactive computer program instead of an inert piece of paper. There appears to be a substantial rate of voter errors when voting on paper ballots—in a Rice University study of paper ballots [24], over 11% of the 126 ballots collected contained at least one error. A friendlier and richer voting interface offered by a computer might help voters avoid making mistakes. Furthermore, the principle of equal rights demands that we provide a way for disabled citizens to cast their votes privately and independently.

* * *

In short, computers can offer several advantages:

- Computers can help manage election-related data.
- Computers can count and tally votes faster.
- Counting by computer avoids human counting errors.
- Computers can offer a richer user interface to voters, potentially improving accessibility and voter accuracy.

Depending on how computers are used in an election, some or all of these advantages may apply.

For this type of election:	Computers could be used to:			
	manage election data	speed up counting	reduce counting error	enrich voting user interface
1. Vote on paper; count by hand.	●			
2. Vote on paper; count by machine.	●	●	●	
3. Vote on machine; count by machine.	●	●	●	●

Figure 1.1. Advantages that computers could *potentially* offer for elections.

How did electronic voting become controversial?

In November 2000, Florida's confusing "butterfly ballot" and heavily disputed punch-card recounts [2, 85] brought highly public embarrassment to the United States election system. The election system suffered widespread criticism on many fronts, particularly for using an outdated counting mechanism. Determined to avoid repeating this fiasco, policymakers and election administrators looked to new technology for a solution. The result was a growing wave of interest in electronic voting, which many hoped would eliminate the ambiguity of punch cards and provide fast, accurate counts.

Two years later, the U. S. Congress passed the Help America Vote Act (HAVA) [78], authorizing hundreds of millions of dollars to be spent on new voting machines. Disability organizations were optimistic about the new requirement for "at least one direct recording electronic voting system or other voting system equipped for individuals with disabilities at each polling place." But computer scientists warned against a hasty switch to electronic voting, citing damage to the transparency and reliability of elections. Though electronic voting machines were already in use in some localities (more than 10% of registered voters used them in 2000 [22]), their adoption surged after HAVA passed in 2002.

In early 2003, election activist Bev Harris made a startling discovery [32]. She used Google to search for "Global Election Systems"—the old name of the company that was acquired by Diebold and renamed "Diebold Election Systems." Diebold Election Systems is one of the heavyweights of the United States election systems industry; its touchscreen voting machine, the AccuVote-TS, was the leading DRE machine used in the 2004 United States election [22]. By following the links from her search results, Harris found a completely unprotected Internet site containing a large collection of company files, including the source code for the AccuVote-TS.

Researchers at Johns Hopkins University and Rice University examined this source code and published a landmark report [43] in May 2004, detailing their discovery of “significant and wide-reaching security vulnerabilities.” They discovered that voters could vote multiple times and perform administrative functions; they found that cryptography was both misused and missing where it should have been used; and they expressed a lack of confidence in the quality of the software in general, concluding that it was “far below even the most minimal security standards applicable in other contexts.” Their findings starkly contradicted Diebold’s public claims that its system was “state-of-the-art,” “reliable,” “accurate,” and “secure” [20].

The state of Maryland then commissioned reviews of the same system from two other agencies: Science Applications International Corporation (SAIC) and RABA Technologies. The SAIC report [72], released in September 2003, confirmed that the system was “at high risk of compromise,” and the RABA report [64], released in January 2004, agreed that the “general lack of security awareness, as reflected in the Diebold code, is a valid and troubling revelation.”

In the 2004 U. S. general election, over 30% of voters cast their votes on electronic voting machines [22]. Voters called in thousands of reports of machine problems, including total breakdowns, incorrectly displayed ballots, premarked choices on the ballot, incorrectly recorded votes, undesired cancellation of ballots or selections, and nonfunctioning or incorrect audio [82].

Since 2004, further investigations have continued to tear down the façade of confidence in the security of voting machines, the claims of vendors, and the testing regime under which the machines were certified. Media story after media story reported on conflicts of interest, regulatory failures, and newly exposed technical vulnerabilities in all the major voting systems, not just Diebold’s.

In the summer of 2007, the California Secretary of State conducted a “top-to-bottom review” of the voting systems used

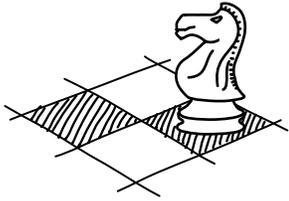
in California, in which I had the opportunity to participate as a reviewer. This was the broadest review of voting system source code to date; the review included source code for DRE machines and optical scan machines from each of three major vendors (Diebold Election Systems, Sequoia Voting Systems, and Hart InterCivic), as well as the election management software responsible for ballot preparation and tallying. However, the review teams only had five weeks to examine the source code. Despite the short time frame, they found serious and pervasive security problems in every system reviewed [7, 12, 35]. The software was not written defensively; security measures were inadequate, misapplied, or poorly implemented; the presence of numerous elementary mistakes suggested that thorough testing had not been done. In particular, every system was found vulnerable to catastrophic viral attacks: the compromise of a single machine during one election could affect results throughout the jurisdiction and potentially affect the results of future elections.

As of this writing, it has become clear that we cannot trust our elections to the electronic voting machines of today's leading vendors. Whether we will ever be able to trust them remains an open question. There is not yet a clear consensus on what standards a voting machine should reasonably be expected to meet. It is also by no means obvious that any set of feasible technical requirements would yield a voting machine worthy of our trust—it might simply be beyond the state of the art to create a sufficiently reliable and economical electronic voting machine. The point of this work is to make progress toward a better design, so as to bring us closer to understanding what is possible and to inform our standards and expectations for these machines.

Why does software correctness matter?

Switching from mechanical to electronic voting machines is a bigger step than it might seem at first. Today's electronic voting machines are not just electrically-powered devices performing the same function as their mechanical predecessors, the way electric light bulbs replaced oil-burning lanterns. Electronic voting machines contain general-purpose digital computers, which makes them fundamentally different and capable of much more than the special-purpose machines they replace. It would really be more accurate to call them "voting computers," as they are called in the Netherlands.

Just like any other general-purpose computer, a voting computer can be programmed to do anything—count votes, miscount votes, lie to voters, play games, or even attack other computers. To prove the point, a Dutch group called "Wij vertrouwen stemcomputers niet" ("We do not trust voting computers") reprogrammed the Nedap ES3B, their nation's leading voting computer, to play a passable game of chess [31].



Consequently, the types of attacks that are possible against voting computers are also fundamentally different than those possible against mechanical voting machines. Tampering with a lever machine can cause it to lose some votes or stop working entirely. Tampering with a computer can cause it to actively engage in sophisticated schemes to deceive voters and pollworkers, behave in different ways at different times or under different circumstances, and even subvert or conspire with other computers.

The behaviour of a general-purpose computer is determined entirely by its software. Assuring the correctness of software has been a major unsolved problem in computer science research for decades. Computer scientists have been able to prove some aspects of correctness for small programs, but all will readily acknowledge that nobody knows a general method for proving software programs to be correct. The software developed in industry tends to be larger and more complex

than can be analyzed by the best known techniques, while the programming languages and tools used in industry generally lag behind the state of the art in research.

Mistakes in software can remain latent for years, even when the code is publicly disclosed and inspected by motivated programmers. For example, OpenSSH is a popular program for secure login. Its developers have declared security to be their number one goal [17], and they have gained a reputation for security practices more rigorous than most. Nonetheless, security flaws were discovered in OpenSSH in 2003 that had been present since its first release in 1999, and had survived intensive software audits by the OpenSSH team.

The problem is exacerbated by the possibility of *insider attacks*: what if someone involved in writing the voting software wants to bias the election? As far as anyone knows, the flaws in OpenSSH were inadvertent mistakes, so intentional flaws can probably be made even harder to find. (Chapter 8 offers some anecdotal evidence that detecting purposely hidden software flaws can be extremely difficult.) Reviewing the voting software is not just a matter of looking for code that seems intended to change votes or tallies. Any flaw that lets an attacker infiltrate the machine is a serious problem, since that flaw can then be exploited to reprogram the machine to do anything. So, a malicious programmer of voting machine software doesn't have to write suspicious-looking vote-altering code; he or she only needs to leave an innocent-looking security weakness. When a security weakness is found, there's no way to tell whether it is an intentional backdoor or an inadvertent mistake—as long as someone knows the flaw, it can be exploited. If any flaw can be an attack, we need voting software to be essentially flawless.

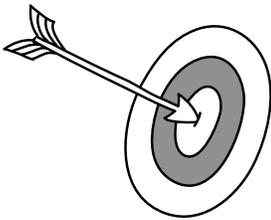
All of this explains why this dissertation focuses on software correctness. There are people who have many years of experience managing election personnel and running paper-based elections. There are people who know how to build reliable machines and reliable computer hardware. But the part that no one fully understands yet is how to get the software right.

2 Correctness

What constitutes a democratic election?	17
What does it mean for a voting system to be correct?	19
How does correctness relate to safety?	20
What is the tree of assurance goals for an election?	24
What does it mean for a voting system to be secure?	30

What constitutes a democratic election?

The democratic ideal of a legitimate election requires that the results reflect an unbiased poll of the voters—*accurate* according to what each voter intended, and *fair* in that each eligible voter has equal and unhindered opportunity to influence the outcome. These two basic goals can be broken down according to the mechanics of how elections are run.



Accuracy. By “accurate,” I mean that the data about voter preferences is accurately gathered and combined to produce the final result. To make this happen, each ballot has to be processed correctly at the three stages of voting:

- **Correct ballot:** Each voter should be presented a ballot with complete and accurate information on the contests for which they are eligible to vote.
- **Cast as intended:** Each voter’s recorded vote should match what the voter intended to cast.
- **Counted as cast:** The calculation that decides the outcome should accurately incorporate every recorded vote and no extraneous votes.



Fairness. By “fair,” I mean that eligible voters (and only eligible voters) are free to vote as they please, without bias. We can look at this from two angles: how the sample of voters is drawn from the population, and how the opinions of the voters are measured.

- **Unbiased sampling:** Votes should come from a fair sample of the population of eligible voters.
- **Unbiased measurement:** Each vote should be a fair measurement of a voter’s preference.

Each of these two aspects of fairness can be elaborated in further detail. In modern democracies, fair sampling is upheld through measures aimed at offering equal access to the polls, and also through the principle of “one person, one vote.”

- **Unbiased sampling** is achieved by ensuring:
 - **Authorized voters:** Only voters that are eligible for a contest should be permitted to vote on it.
 - **One ballot per voter:** No voter may cast more than one ballot.
 - **Equal suffrage:** Every voter eligible for a contest should have an equitable opportunity to vote on it.

An unbiased measurement depends on eliminating influence from external pressures as well as influence from the presentation of the ballot itself.

- **Unbiased measurement** is achieved by ensuring:
 - **Secret ballot:** No voter's choices should be exposed by the voting system or demonstrable by the voter to others, lest votes be influenced by social pressure, bribery, threats, or other means.
 - **Equal choice:** Every option in a contest should have an equitable opportunity to receive votes.

Democracy also demands a further virtue: since power is derived from the consent of the governed, the election process itself must be accountable to the people. The manner in which all of the above goals are achieved should be **verifiable**, so that members of the public can assure for themselves that the election is accurate and fair. The verifiability of the election is not listed among the above goals because it is a “meta-goal,” like a layer on top of all the other goals.

A widely preferred avenue for achieving verifiability is through transparency—exposing the election process to public scrutiny. However, verification can also take place through the investment of trust in independent experts or inspectors (or suitably balanced committees thereof), or through cryptographic means, in which a calculation provides mathematical evidence of the property to be verified.

What does it mean for a voting system to be correct?

In order to be confident that an election is democratic, we would want to have assurance of all of the goals just mentioned. But these goals are for the election as a whole, including all the people, processes, and technology involved. When we talk about a particular piece of equipment, such as a voting machine, we have to choose a specific set of subgoals that it is responsible for. For example, a voting machine cannot, by itself, guarantee that each voter only votes once. However, if the machine requires something like an access card in order to cast each ballot, this feature *in combination* with a suitably controlled process for handing out access cards, carried out by competent, trustworthy pollworkers, can effectively limit each voter to casting just one ballot.

Every goal is achieved through some combination of human processes and technology. This dissertation is primarily concerned with the technological part of an election—the equipment and software involved in collecting and counting votes, which I am calling the “voting system” for short. To say that the voting system works correctly means that it fulfills the responsibilities that have been assigned to it. Only after we’ve decided on this assignment of responsibilities is it meaningful to say whether it is correct. As the access card example illustrates, it is usually necessary to subdivide goals in some detail in order to separate out subgoals that technology can address.

How does correctness relate to safety?

Engineers have been designing safety-critical systems for many years, so it's instructive to examine the research and practice in methodologies for developing these systems.

Analysis. One of the most common analysis techniques for safety-critical systems is *fault tree analysis* [83]. Fault tree analysis is a way of identifying all the ways that a particular failure can occur. To perform fault tree analysis, one begins with a root node that represents the undesired event (the fault); then one identifies all the events or situations that could cause that undesired event, and each one becomes a child of the root node. Each node can be further refined by adding children that identify possible causes. For example, a few nodes in a fault tree for a fire extinguisher might look like this:

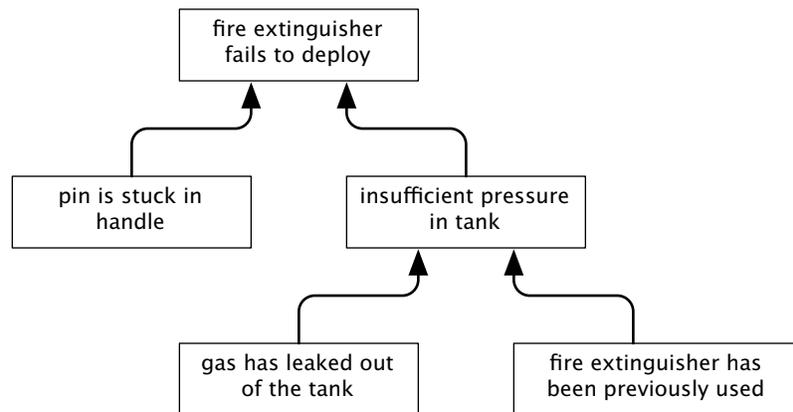


Figure 2.1. A small portion of a fault tree for a fire extinguisher.

Fault trees are known in the computer security world as *threat trees* [3] or *attack trees* [70]. An attack tree lays out all the possible ways that an attacker might come to violate a specific security restriction. In an attack tree, the top node is the attacker's ultimate goal. The children of a node specify various ways that an attacker can achieve the goal. For example, if the ultimate goal is to break open a safe, an attacker could do

so by obtaining the combination or by drilling open the safe. Part of the attack tree might look like this:

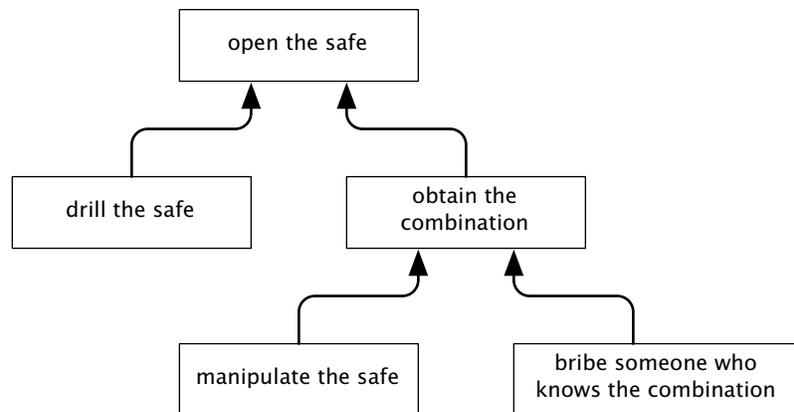


Figure 2.2. A small portion of an attack tree for an attacker who wants to break into a safe.

In the above examples, any one of the children of a node is sufficient to lead to the parent; the relationship among siblings is a disjunction (OR). Fault trees and attack trees can also specify conjunctions (AND) and other logical relationships. The nodes can be labelled with numbers to indicate the probability of an event or the cost of a step in an attack.

Design. Fault trees and attack trees are used to analyze existing systems to identify their weaknesses. But when one is designing a system, the goal is to establish the system's worthiness.

In the safety-critical literature, a written justification of a system's safety is called a *safety case* [87]. Safety cases are required by many safety standards. A safety case is often a very large document, as it incorporates all the arguments and supporting evidence for the safety of each element of the system. The development of the safety case can take up a large fraction of the effort in designing a safety-critical system. Hence, significant research efforts have been directed toward ways of organizing and maintaining safety cases.

Like fault trees, safety cases are also typically structured in a top-down approach based on successive refinement. The technique that is probably the most prominent in the research

literature is the Goal Structuring Notation [41], which elaborates on a basic tree-like organization of goals by allowing nodes of several different types: goals, strategies, justifications, assumptions, and so on. Here is an example of a section of a safety case for a microwave oven diagrammed in Goal Structuring Notation:

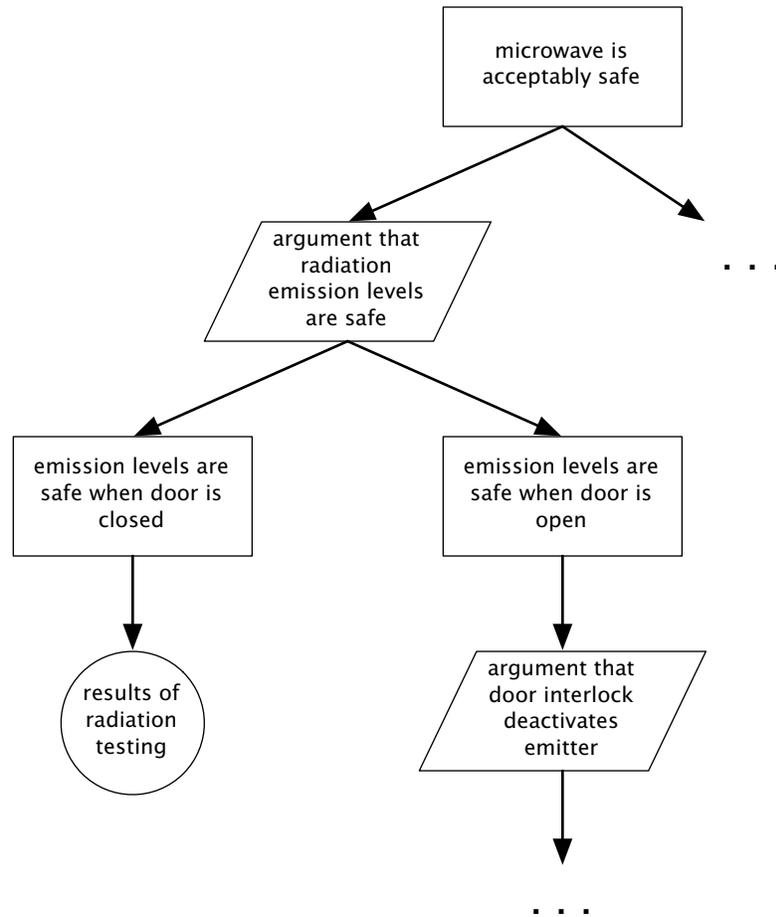


Figure 2.3. Part of a safety case for a microwave oven in Goal Structuring Notation.

Voting systems. A safety case would be appropriate for justifying why we should place our confidence in a voting system. Ideally, certification of any voting system for deployment would require the manufacturer to provide a convincing and clearly structured safety case.

The hierarchy of goals for a democratic election form the

starting point for such a safety case. The process of dividing goals into subgoals produces a tree of assurance goals for a system, which I'll call an *assurance tree*. (An assurance tree could be considered a simplified instance of Goal Structuring Notation in which all the nodes are correctness goals.) When an assurance tree is fully elaborated, the leaves of the tree are individual responsibilities that can be assigned to specific people and specific devices.

The process of refining the general goals into specific subgoals is a type of design activity. Different solutions will subdivide the main goals differently and assign responsibilities for the subgoals differently. For example, access cards are one possible way to keep voters from voting multiple times, but of course they are not the only way. It is a design choice to implement “one ballot per voter” in terms of the two parts: “pollworkers give one access card to each eligible voter” and “the voting machine allows each access card to be used just once to cast a ballot.” Making these design choices and refining the goals at every level eventually leads to a set of specific technical requirements for the voting system.

In an assurance tree, the children of each node indicate what requirements have to be upheld in order for the parent goal to be upheld. The final result of refining the tree is an assignment of specific responsibilities to various parts of the system—for example, a set of tasks to be carried out by humans and a set of tasks to be carried out by computers—such that all the assurance goals are upheld. The tree captures the design of the system as well as the security assumptions that the designer made.

What is the tree of assurance goals for an election?

The requirements that were presented earlier can be refined one step further without specifying a particular voting system design. First I'll explain each subgoal, then present the whole tree, which can form a basis for the safety case of any election.

Accuracy: correct ballot. In order for a voter to receive a correct ballot, the correct ballot has to exist for that voter and it must contain the correct instructions and choices for the election. The voter then has to be given the right kind of ballot, and the voter has to receive it without alteration.

Accuracy: cast as intended. The voter's vote is properly recorded if the ballot indicates *what* the voter wanted and is cast *when* the voter is ready. Choices should be selected if and only if the voter makes them, and the voter should be free to mark the ballot in any manner that is valid. (When paper is used, the voter can also cast an invalid ballot; then the ballot is not counted. When electronic machines are used, the machine usually prohibits the voter from marking the ballot in an invalid manner.) To further ensure that the cast ballot matched the voter intent, the voter should get accurate feedback about what is currently selected, and should be able to make changes or corrections before casting the ballot.

Accuracy: counted as cast. For the count to be correct, there must be no extra or missing votes, and the votes that are counted must be exactly as voters indicated them on their cast ballots.

Fairness: authorized voters. I use the term *voting session* for the interval that begins with a voter entering a protected area of the polling place such as a voting booth, and ends when the voter walks away, either having cast or failed to cast a ballot. In

a typical election, voter authorization consists of controlling access to voting sessions and ensuring that there is no other way to cast a ballot except in a voting session.

Fairness: one ballot per voter. Limiting each voter to one cast ballot is also achieved by controlling access to voting sessions. In practice, each voter is authorized for one voting session at a time. If a voter wants to try again, a pollworker either destroys the ballot or determines that the voter did not already cast a ballot, and then authorizes another voting session.

Fairness: equal suffrage. There are three steps to casting a ballot. First the voter has to get to a polling station. Then, at the polling station, the voter has to be allowed to begin a voting session. Then, in the voting session, the voter has to successfully cast the ballot. Equal suffrage demands that voters have reasonable access and be free of discrimination at all of these stages.

Another way that a voter can be disenfranchised is to make an error. It is infeasible to demand that there be no errors at all, but fairness requires that errors not be biased against any particular group of voters. The controversy over the 2006 race for Florida's Congressional District 13 highlighted the significance of biased error. Different voters saw different ballot layouts, and post-election analysis [29] has suggested that the particular layout used in Sarasota County caused a large fraction of voters to skip the congressional race by mistake.

Fairness: secret ballot. The election system should not itself violate the voter's privacy. But it's a tougher task to prevent coercion. Voters' susceptibility to influence may not be based in reality: as long as voters *believe* they will profit or suffer by voting a certain way, the belief is sufficient to influence their votes. For example, an attacker could claim to have insider access that allows him to identify which voters voted for a particular candidate and punish them. Whether or not the attacker has such insider access, or whether discovering voters'

identities is even possible, the fear of punishment could be enough to sway votes. Different kinds of voting systems will lend differing degrees of plausibility to such claims—for example, some voters might be easily persuaded that someone could violate their privacy via computerized vote records, but they might find it harder to see how such a violation would be possible with hand-counted paper ballots.

The formal definitions of coercion-resistance in the research literature [18, 38, 60] require that voters be unable to *prove* to a vote-buyer that they voted a certain way. But the issue is more nuanced than that. A vote-buyer doesn't need solid proof, just evidence sufficiently plausible that offering a reward for it will influence the vote.

For example, consider an election system in which voters receive receipts indicating how they voted, but could also *forg*e such receipts. One might think that such an election system is coercion-resistant, since it isn't worthwhile for a vote-buyer to buy something that can be forged. But resistance to coercion also depends on the cost of producing a forgery: if forgeries require enough effort that a significant number of voters will vote as directed by the vote-buyer instead of carrying out the forgery, the vote-buyer will succeed at influencing the election. Therefore, the secret ballot goal includes the requirement that voters not be given any *plausible evidence* (not just hard proof) of their votes that could be sold to an external party.

Fairness: equal choice. Since the goal is to avoid bias among the options within a contest, it would not do for some of the options to be shown one way to some voters and a different way (say, in red, or in larger print) to others.

It would be ideal to avoid all bias among options presented on the same ballot, but this is not possible: some option has to be presented first, and there is a well-documented bias toward the first item [46]. The next best thing is to change the order of presentation from ballot to ballot such that there is a uniform *distribution* of bias towards all the options, when the ballots are considered in aggregate.

There is a more subtle kind of bias that also should be avoided: a bias relative to the voter's preferred choice. Imagine, for example, a contest with three options A, B, and C. Suppose the ballot design causes half the voters who intend to mark A to mistakenly mark B, half of those who want B to mark C, and half of those who want C to mark A. Such a ballot is not biased toward any particular option, but it is still clearly unfair: B could win an election in which most voters intended to vote for A. So there is also a requirement for a uniform distribution of errors with respect to the voter's intended choice.

* * *

Gathering all the requirements just mentioned gives us the following high-level assurance tree for elections.

Accuracy

- **Correct ballot**

- G1. For every voter, there exists a ballot style containing the complete set of contests in which that voter is eligible to vote.
- G2. On every ballot, all the information is complete and accurate, including instructions, contests, and options.
- G3. In every voting session, the correct choice of ballot style is presented to the voter.
- G4. Every ballot is presented to the voter as the ballot designer intended.

- **Cast as intended**

- G5. At the start of every voting session, no choices are selected.
- G6. The voter's selections change only in accordance with the voter's intentions.
- G7. The voter receives accurate feedback about which choices are selected.
- G8. The voter can achieve any combination of selections that is allowable to cast, and no others.

G9. The voter has adequate opportunity to review the ballot and make changes before casting it.

G10. The ballot is cast when and only when the voter intends to cast it.

- **Counted as cast**

G11. Every selection recorded on a ballot cast by a voter is counted.

G12. No extra ballots or selections are added to the count.

G13. The selections on the ballots are not altered between the time they are cast and the time they are counted.

G14. The tally is a correct count of the voters' selections.

Fairness

- **Unbiased sampling**

- **Authorized voters**

G15. Only authorized voters can begin voting sessions.

G16. Only in voting sessions can ballots be cast.

- **One ballot per voter**

G17. No voting session allows more than one ballot to be cast.

G18. Each voter is allowed at most one voting session in which a ballot was cast.

- **Equal suffrage**

G19. Every voter has reasonable, non-discriminatory access to a polling station they can use.

G20. Every voter can begin a voting session within a reasonable, non-discriminatory waiting time.

G21. Every voting session provides a reasonable, non-discriminatory opportunity to cast a ballot.

G22. For every voter that is eligible to vote in a particular contest, there is a uniform likelihood of voter error on that contest.

- **Unbiased measurement**

- **Secret ballot**

G23. The processing of voter choices does not expose how any particular voter voted.

G24. Voters are not provided any way to give plausible evidence of how they voted to an external party.

□ **Equal choice**

G25. Within each contest, all the options are presented in the same manner on each ballot and across all ballots.

G26. For each contest, the voters are presented with ballots that, in aggregate, yield a uniform distribution of bias in favour of each option.

G27. For each contest, the voters are presented with ballots that, in aggregate, yield a uniform frequency of voting errors across the voters that intend to vote for each option.

G28. In each contest, for each option, voters intending to vote for that option are presented with ballots that, in aggregate, yield a uniform distribution of voting errors in favour of every other option.

What does it mean for a voting system to be secure?

A voting system is secure if it can be relied upon to produce the correct results in the face of determined attempts to corrupt the outcome. Thus, security and correctness are closely related: security is just correctness in an adversarial context. The intentional violation of any subgoal in the assurance tree would constitute a security breach.

Since this dissertation is focused on the software security questions surrounding electronic voting machines, let's separate out the goals that rely on software from those that don't.

Of the goals in the assurance tree, these are normally addressed by humans in the preparation and conduct of the election:

- G1. For every voter, there exists a ballot style containing the complete set of contests in which that voter is eligible to vote.
- G2. On every ballot, all the information is complete and accurate, including instructions, contests, and options.
- G18. Each voter is allowed at most one voting session in which a ballot was cast.
- G19. Every voter has reasonable, non-discriminatory access to a polling station they can use.

The following goals are addressed through good ballot design. They could be violated by voting machine software that displays the ballot incorrectly or lacks the ability to display ballots in a fair manner. However, as long as the voting machine presents the ballot as the ballot designers intended (which is goal G4), we can consider these goals the responsibility of ballot designers:

- G22. For every voter that is eligible to vote in a particular contest, there is a uniform likelihood of voter error on that contest.
- G25. Within each contest, all the options are presented in the same manner on each ballot and across all ballots.

- G26. For each contest, the voters are presented with ballots that, in aggregate, yield a uniform distribution of bias in favour of each option.
- G27. For each contest, the voters are presented with ballots that, in aggregate, yield a uniform frequency of voting errors across the voters that intend to vote for each option.
- G28. In each contest, for each option, voters intending to vote for that option are presented with ballots that, in aggregate, yield a uniform distribution of voting errors in favour of every other option.

The following goals could be addressed almost entirely by election-day procedures, or through a combination of such procedures and proper software behaviour, depending on how the voting system is designed:

- G15. Only authorized voters can begin voting sessions.
- G16. Only in voting sessions can ballots be cast.

The proposed designs in this dissertation assume that the above two goals are upheld by human procedures. For G15, election workers ensure that only authorized voters are permitted physical access to voting machines. And for G16, election workers should provide no other way to cast ballots outside of the officially approved procedures.

The remaining goals are those that necessarily depend on the correctness of the voting machine software implementation:

- G3. In every voting session, the correct choice of ballot style is presented to the voter.
- G4. Every ballot is presented to the voter as the ballot designer intended.
- G5. At the start of every voting session, no choices are selected.
- G6. The voter's selections change only in accordance with the voter's intentions.
- G7. The voter receives accurate feedback about which choices are selected.

- G8. The voter can achieve any combination of selections that is allowable to cast, and no others.
 - G9. The voter has adequate opportunity to review the ballot and make changes before casting it.
 - G10. The ballot is cast when and only when the voter intends to cast it.
 - G11. Every selection recorded on a ballot cast by a voter is counted.
 - G12. No extra ballots or selections are added to the count.
 - G13. The selections on the ballots are not altered between the time they are cast and the time they are counted.
 - G14. The tally is a correct count of the voters' selections.
 - G17. No voting session allows more than one ballot to be cast.
 - G20. Every voter can begin a voting session within a reasonable, non-discriminatory waiting time.
 - G21. Every voting session provides a reasonable, non-discriminatory opportunity to cast a ballot.
 - G23. The processing of voter choices does not expose how any particular voter voted.
 - G24. Voters are not provided any way to give plausible evidence of how they voted to an external party.
- G3 and G20 depend on election-day procedures as well as the voting machine software. For G3, typically a pollworker is responsible for selecting the correct ballot style for each voter, and the voting machine must correctly use the ballot style indicated by the pollworker. For G20, the polling station needs to serve voters efficiently and fairly, but also the voting machines should be available and ready to serve voters and should not freeze up or crash. G23 and G24 depend on the overall design of the voting system, including the human procedures, as well as the correct functioning of the voting machine software.

Security issues with voting machine software usually have to do with upholding and enforcing the 17 goals in this last list. These 17 goals are the focus of my efforts to achieve and verify software correctness.

3 Verification

How do we gain confidence in election results?	34
How can we verify the computerized parts of an election?	36
What kind of election data can be published?	39
What makes software hard to verify?	41
In what ways are today's voting systems verifiable?	44
What is the minimum software that needs to be verified?	48
What other alternatives for verification are possible?	52

How do we gain confidence in election results?

An election consists of many steps, each of which processes information such as ballot and candidate data, voter information, and records of cast votes. At the most basic level, each step takes some input and produces some output. Confidence in the ultimate result—the output of the last step in the chain—depends on confidence that each step was correctly performed. The choice of the type of voting system determines which steps are carried out by people and which by computers.

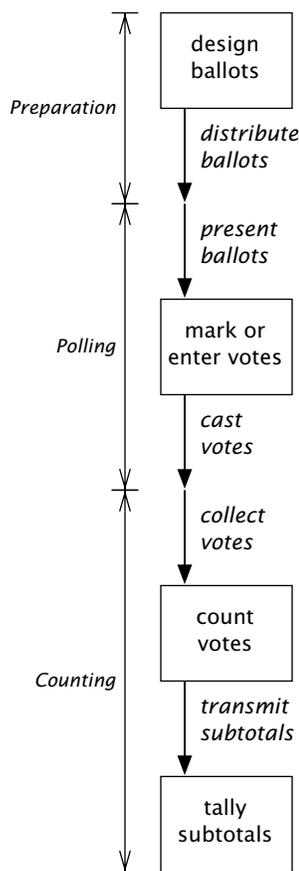
Earlier we described the election process in terms of three stages: preparation, polling, and counting. With respect to establishing confidence in a voting system, these stages can be broken down further into the nine steps shown at the left, which include transmission as well as processing of information.

The *preparation* stage consists of events prior to the opening of polls, which includes not only designing the ballots but also distributing them to polling places. This production and distribution takes place for both paper ballots and electronic ballot definition files.

The *polling* stage involves presenting the ballots to voters, who make selections and cast the ballots. For sighted voters reading paper ballots, presentation of the ballot is a trivial step, but for electronic voting computers the fidelity of the presentation is a real issue.

In many elections, *counting* occurs in two parts: votes are first counted at polling places, then the counts are centrally tallied to yield the final results. This stage includes the transmission of votes to the person or machine that counts them. The distinction between local and central counting is important because the local counting process often takes place in public, whereas the aggregation of results and central tallying does not.

For a step that transforms information from one form to another, confidence comes from ensuring that it produced the correct output for the input it was given. For a step that



transports information from one place to another, confidence comes from ensuring that the integrity of the information was preserved.

Because of the way I've defined the three accuracy goals (*correct ballot*, *cast as intended*, and *counted as cast*), they differ slightly from the three chronological stages: getting the correct ballot to the voter includes the presentation step at the polls. The following figure shows which steps correspond to the three accuracy goals. Under each step is the name of a subgoal for that step.

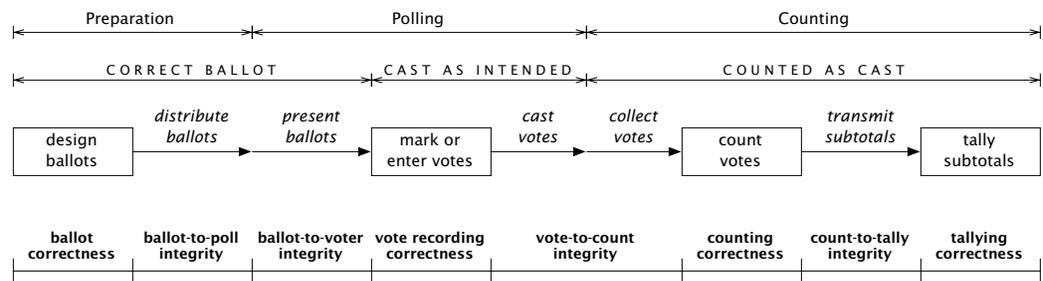


Figure 3.1. The nine steps in the election process and their corresponding integrity and correctness goals.

How can we verify the computerized parts of an election?

Suppose that a particular information processing step in an election is carried out by a computer. As I mentioned in Chapter 1, the computer's behaviour is completely controlled by its software. Let's say the software program responsible for this step takes some input x and produces some output y . For example, if this is the vote-tallying step, x could be a collection of electronic vote records and y could be the election totals.



Figure 3.2. For some particular processing step in an election, a software program takes the input x and produces the output y .

If you want to check that the program produced the correct result, you have two main choices:

1. *Software verification.* You can examine the program itself and confirm that it works the way you expected. Depending on the assumptions you make, this may include manual inspection of the source code, automated analysis, or formal mathematical proofs. Once you have confirmed that the program does exactly what it's supposed to do in every possible circumstance, you can be confident that this particular output, y , is correct.
2. *Result verification.* You can take the input x and figure out what the corresponding output should be. If the actual output y matches the expected output, then you know it's correct. To do this, you need records of both x and y , as well as some way to independently repeat the operation—perhaps you have another program that you trust, or perhaps you can work out the expected output by hand.

There is also a variant of result verification:

2a. *Indirect result verification.* Some schemes allow you to establish confidence without repeating the entire operation. For example, given information *derived* from x and y , you might have a way to mathematically check their consistency. Or, you might be allowed to choose *parts* of x and y to check, enabling you to establish a high *probability* of a correct result.

Software verification has the advantage that it only needs to be done once on a given program to establish confidence in all the output it will ever produce. Result verification has to be repeated each time the program produces new output. However, there are three major factors weighing in favour of result verification.

Programs change. The apparent advantage of doing software verification only once becomes less compelling when you consider that software changes all the time. Features are added; bugs are discovered and fixed; demands change. In particular, election software is subject to election law, which differs from state to state in the United States. Whenever legislation gets passed, election software may have to be updated to satisfy new requirements. Any change would invalidate previous reviews or proofs of correctness and require the software to be verified over again.

Software verification requires disclosure. Disclosure of software code often faces legal, financial, or political barriers. Voting machine companies have resisted public disclosure of their source code on the grounds that it could help a motivated attacker, and they claim that copyright and trade secret protection are necessary to support a sustainable, profitable business. [34] Disclosing code would certainly increase the transparency of an election and improve the accountability of the testing process. But having ways to check the correctness of an election without *depending* on disclosure of all the code would allow the election to sidestep this disclosure dispute. The

democratic process is healthier if private interests have fewer opportunities and fewer plausible incentives to prevent the public from verifying an election.

Software verification is much harder. As a later section of this chapter will explain (page 41), the behaviour of software can be extremely difficult to analyze. Software review by human experts is expensive, time-consuming, and prone to error. The only way to be truly sure is to construct a mathematical proof, but it is well beyond the state of the art to do this for programs the size of typical computer applications. When such proofs are constructed, they often aim to prove things about a simplified model of the program rather than the program itself. Unfortunately, a mathematical proof can only prove that a program satisfies a formal specification of what it's supposed to do. The proof only establishes that the program is correct if the specification accurately expresses what it means to be correct—and such specifications are themselves complex and tricky to write.

What kind of election data can be published?

There is an inherent tension between voter privacy and the desire for verifiable elections. As argued earlier in this chapter, verifying results is preferable to verifying software. But public verification of results depends on publishing election data.

Suppose there is some data made available to the public to enable verification. This might include partial or complete information about ballots, votes, and results, or something derived from such data. Each published piece of data (let's call it a *record*) might be *identifiable* as corresponding to a particular voter, or it might not. And each record might contain sufficient information to *reveal votes*, or it might not. These two features are independent: for example, a published record could indicate a vote for a particular candidate, yet not be associated with any particular voter.

For voters to be able to check that their own ballot was correctly received (i.e., cast as intended), they need to be able to look up their own ballots. To do this, they need some kind of public record of their ballot that is *identifiable*.

For voters to be able to confirm the tally by directly performing their own recount, they have to be able to see the votes. To do this, they need public records that *reveal votes*.

Published records that are identifiable *and* reveal votes would enable the public to verify everything, at the expense of voter privacy. Imagine an election in which every ballot is published online and uniquely associated with the voter who cast it. Any voter could look up their ballot online to confirm that it is correct as published, and anyone could count the published ballots to confirm the tally. In such a system, software correctness would be irrelevant—software could be used at any stage of the process and there would be no need to verify it, because the entire election can be checked by result verification. But in such an election, voters could also easily sell their votes—for example, they could tell a vote-buyer where to find their ballots online.

* * *

In summary:

- Public confirmation that ballots are cast as intended requires public records that are *identifiable*.
- Public confirmation of the tally by direct recount requires public records that are *vote-revealing*.
- If any public records are identifiable *and* vote-revealing, they enable bribery and coercion.

This suggests two possible kinds of public records:

1. Anonymous records that do reveal votes.
2. Identifiable records that don't reveal votes.

Several proposals for voting systems, including those proposed in this dissertation, publish records of the first kind. These records enable direct result verification of the tally. Later in this chapter, I'll discuss end-to-end cryptographic voting systems, in which both kinds of records are published, and an additional verification step confirms the correspondence between the two.

What makes software hard to verify?

Most software is hard to verify because it is complex.

Here are some of the main reasons why complexity in software is more difficult to manage than complexity in a physical machine.

Number of components. The number of parts in a physical machine is limited by the costs of manufacturing, but there is no such limit on software. A software program costs the same to distribute—virtually nothing—whether it contains ten components or a million components. It is easier to add complexity to a software program than to a physical device, and removing code often has a higher risk of breaking the program than adding new code. Requirements change and customers ask for more features; in response, software tends to grow boundlessly during the course of development, unless there are determined and persistent efforts to keep it small.

Software programs also often incorporate large ready-made packages of components written by others, to save the effort of writing code from scratch. Even if only a small part of a package's functionality is used, it is easier to include the entire package than to separate the parts that are used from those that are not. These pressures lead to software applications with millions of lines of code and thousands of interacting components.

Complex interconnections. There are likely to be more connections between the parts of a software program than those of a physical machine. Whereas a machine part can only interact with other parts near it, there is no limit on the number of other parts that a software component can depend on. For example, it is common for a single component to be relied upon by thousands of other components.

These connections are also harder to see in software. The way that a machine part affects other parts is usually clear from

direct physical inspection. But finding all the other software components that depend upon a given software component can be a difficult task.

Far-reaching effects. Because software components can be so deeply interconnected, a small change in one part can affect another part that is far away, affect parts written by different people, or have wide-ranging effects on the behaviour of the whole program. The software engineering practices of modularity (dividing up a program into distinct modules) and encapsulation (protecting each module from outside interference) aim to limit these kinds of effects, but software programs nonetheless tend to be more sensitive to change than physical machines.

Nonlinearity. The power of general-purpose computers derives from their ability to make decisions. With software, a tiny change in input can yield a completely different outcome; for example, a program can decide to behave one way when the result of a calculation turns out to be zero and another way when it is nonzero. This means that similar situations cannot be assumed to yield similar behaviour. This *nonlinear* nature makes it hard to predict how software will behave and hard to test software thoroughly. Mechanical devices can be nonlinear too, but software tends to be pervasively nonlinear.

* * *

One of the most serious threats that is currently poorly addressed in voting systems is the insider threat from software developers. Intentionally placed bugs or backdoors are hard to detect even when software is carefully audited [5]. The persistent failure of the federal testing process to detect major security flaws [21, 37] and the continuing revelations of security vulnerabilities in certified voting systems [33, 43, 64, 84, 88] suggest that voting software has not been audited anywhere near enough to defend against this threat.

The complexity of software is what makes it difficult to be *sure*: sure that the software will behave as expected, that it will produce the correct results, and that it will resist determined attempts to subvert the outcome of an election. Software complexity is the ultimate enemy of reliable computer-based elections.

There are two ways to fight this enemy: design the system so less of the software needs to be verified, and simplify the software that needs to be verified. Both can be applied together.

In what ways are today's voting systems verifiable?

Different voting systems offer different ways for voters to gain confidence that the election results are correct. We can compare systems by looking at what mechanism for assurance is provided, if any, at each step of the process.

The two kinds of voting technology most commonly used in the United States are *optical scan* systems and *direct recording electronic* (DRE) systems.

Optical scan voting. When an election is conducted by optical scan, paper ballots are prepared and printed before polls open. Voters mark the ballots by hand and deposit them into a ballot box. There are two variants of optical scan voting: the scanning can take place at individual precincts or at a central election office.

Although software is usually involved in preparing the ballots, voters and candidates can verify for themselves the sample ballots published before polling. Voters can also bring sample ballots to the polling place and compare them with the blank ballots they receive. This is an example of avoiding software verification, which is possible because the results of the preparation stage are public.

We know the ballot is presented exactly as prepared, because the voter directly reads the printed paper. There is no recording device to misrecord the voter's marks; the voter is responsible for clearly marking the paper to be counted. The election relies on the physical durability of paper for the integrity of printed ballots and recorded votes.



A precinct-based optical scanner.

When scanning takes place at individual precincts, the ballots pass through a scanning machine on their way into the ballot box. After polls close, each machine prints out its counts on a paper tape. If the paper tapes are posted immediately for public viewing, then no one has to trust the software that does the tallying. The final election report will contain both the

counts in each precinct and the overall totals. Anyone can confirm that the locally posted results are correctly included in the election report, and anyone can confirm that the overall totals were calculated properly.

When scanning is performed centrally, voters can't perform the same check on the tally step. They have to trust election personnel to safely transport the ballots from the polls to the central office and to enter the results from the central scanner into the software that tallies them (known as the election management system, or EMS).

Figure 3.3 summarizes the mechanisms by which any individual voter can ensure the validity of each step in this process. (I'll call this an *assurance chart*.)

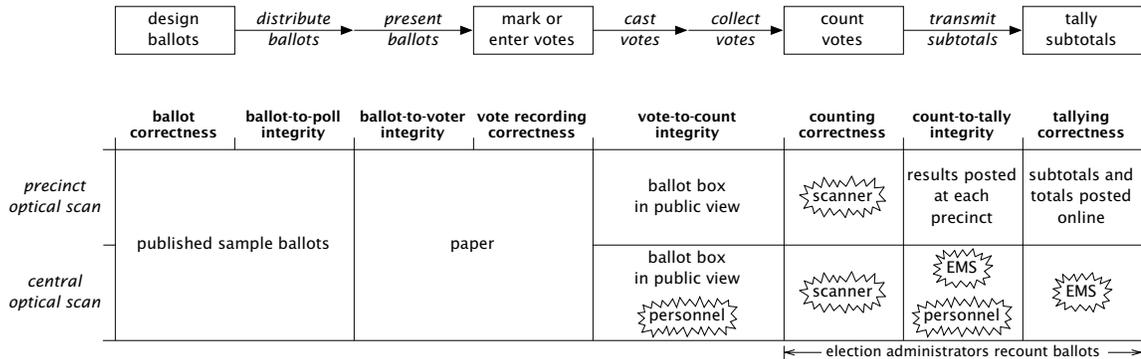


Figure 3.3. Assurance chart for elections with hand-marked, optically scanned ballots.

The starbursts mark mechanisms that voters have to accept on faith—they have to trust software they can't see or people they don't know. For precinct-based scanning, voters have to trust the software that controls the optical scanner. For central scanning, the voters also have to trust the personnel who collect the ballots and convey counts from the scanner to the EMS. They also have to trust the EMS itself, since they have no way to independently check that the totals were added up correctly.

Paper ballots provide a useful backup record, as they can be recounted by hand or by machine. The same stack of ballots can even be counted multiple times, and the counts from different people or different machines can be compared to improve

confidence. In Figure 3.3, recounts are shown as a secondary assurance mechanism, below the three boxes on the right. They are shown as secondary because ordinary voters cannot conduct or order recounts; only election administrators can do so.

DRE voting. Figure 3.4 shows what voters have to trust for each step of an election process with a DRE voting system. There are two possibilities here as well: the results from DRE machines might be reported at each precinct, or they might be reported only by the central election office.

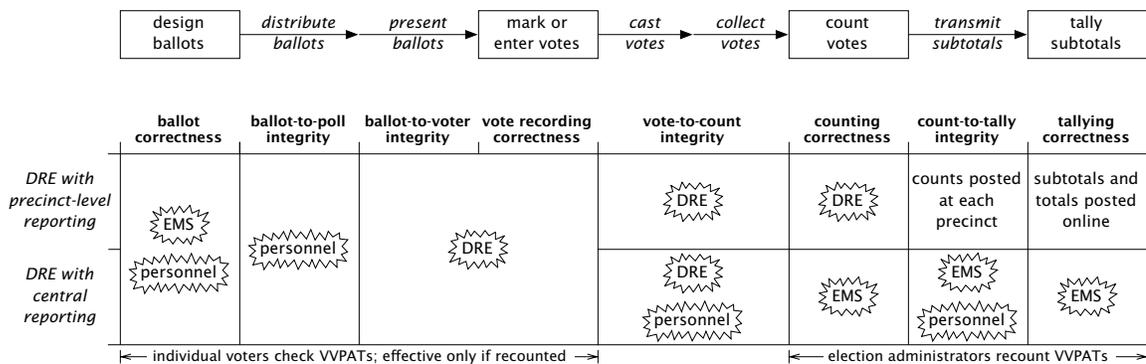


Figure 3.4. Assurance chart for elections with direct recording electronic (DRE) voting.

When DRE machines are used, voters don't get to see a sample of the ballot definition in the machine, in the same way that a sample ballot is a direct preview of what will be used on election day. At best, voters might get images of the screens displayed by the DRE, printed on paper. But, in general, they don't get to test-drive a DRE with the ballot definition they will be using, and they can't check whether their machines have received the correct ballot definitions. Voters have to trust the EMS, which produces the ballot definition files, the personnel that operated the EMS, and the personnel that loaded the ballot definitions into the DRE machines.



A DRE voting machine.

The DRE machines are responsible for presenting the choices to the voter and recording the voter's selections. For these steps the voter is forced to trust that the DRE software is correct. For the counting stage, voters have to trust either the

software in the DRE that counts and reports results locally, or the software in the EMS that counts and tallies the results centrally, along with the personnel that convey the information to the EMS.



A DRE with a VVPAT printer (at lower right).

As a backup verification mechanism, some DRE machines print *voter-verified paper audit trails* (VVPATs). This is a paper tape that shows the voter’s selections for viewing and confirmation by the voter. Printed VVPATs are retained by the machine so that they can later be recounted if a recount is deemed necessary. However, voter inspection of VVPATs is not as strong a backup as voter inspection of paper ballots; in the case of VVPATs, the thing being inspected is not what is normally counted. With DRE machines, the results are derived from the electronic records, not the VVPATs that voters see; the VVPATs are only relevant if election officials decide to conduct a recount.

There are also good reasons to believe that voters are unlikely to catch discrepancies on VVPATs. In a study by Everett [25], voters using a mock DRE were shown a review screen with selections different from what they had chosen, and 68% of voters failed to notice the changes. It seems likely that even more voters would miss discrepancies on the VVPAT, which is generally smaller than the screen and shown off to the side of the machine.

As Figure 3.4 makes obvious, DRE voting systems depend heavily on software. Because so little information is typically published about these programs and their inputs and outputs, trusting the outcome of such an election often requires trusting virtually every piece of software in the system—software for designing ballots, software that produces ballot definitions, voting machine software, software that tallies votes, and all the operating systems, compilers, editors, and other tools that were used to produce these programs.

It doesn’t have to be this way. By publishing information about the software and the data processed by that software, it’s possible to reduce what voters have to accept on faith in order to trust the validity of the election result.

What is the minimum software that needs to be verified?

The degree to which software verification is avoidable depends on a critical decision: how do voters indicate their votes—on paper or on a computer? Of all the steps in the process, this one is special because it must take place in private.

A big part of the present controversy over electronic voting machines is a conflict about the user interface presented to voters. Proponents of the machines point to the real benefits that computers could offer in improved usability and accessibility. For people with certain disabilities, voting computers may be the only way to vote privately and independently. Whether these advantages are enough to outweigh the loss of a tangible, directly marked ballot is a complicated question, and I argue for neither side of that issue here. But an important factor in deciding whether vote entry should occur on paper or on a computer is the feasibility of ensuring the integrity of votes in either case.

Each of the two cases has its own answer to “what is the minimum software that needs to be verified?”

Case 1: The paper option. If voters directly mark paper ballots, the answer is “nothing.” To avoid all software verification, just publicly count the ballots by hand right after the polls close. Sample ballots, mailed out before polls open, let voters check that the real ballots are printed correctly. There is no software involved in marking and casting votes, only paper. And if the results of the hand count are posted immediately at the polling place, then no one has to trust the software that does the tallying.

So, in a voting system where paper ballots are hand-marked and hand-counted at the polls, any step that uses software can be publicly checked by direct result verification. As with any paper ballot system, the ballots are available to be recounted later if necessary.

Figure 3.5 summarizes the preceding analysis in an assurance chart.

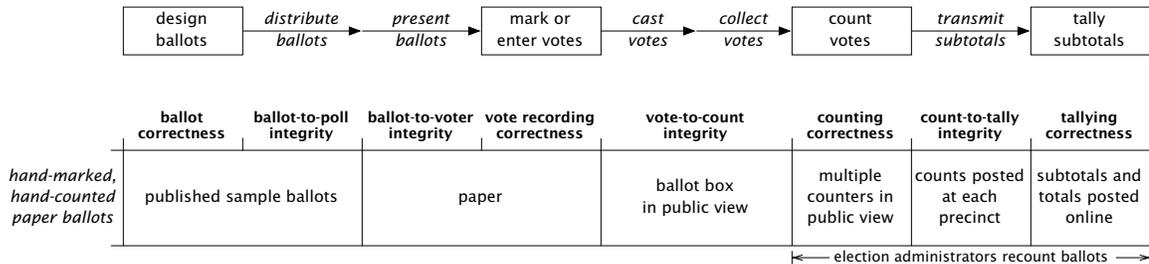


Figure 3.5. Assurance chart for an election with hand-marked, hand-counted ballots.

Case 2. Entering votes by computer. In this case, the answer is “just the vote-entry software.” Here’s why.

The “mark or enter votes” step, central to the voter experience, also turns out to be critical in terms of verification. This step cannot be publicly verified by result verification. Result verification requires a complete record of inputs and outputs. But one of the inputs to this step is the input from individual voters, which must be kept private due to the principle of the secret ballot. Moreover, if the ballot is presented to the voter by a computer, the voter’s input is subject to influence by the computer.

Therefore, if choices are presented or selected on a computer, software verification is unavoidable. However, the secret ballot is the only privacy requirement that elections have to uphold. Recorded votes can be published as long as they cannot be associated with any particular voter. **The only part of the process that needs to be secret—and thus the only part for which software verification is really necessary—is from the private interaction with an individual voter up until the moment the voter’s votes are recorded in anonymous form.** That interval is the critical interval during which private information gets turned into publishable information. All the inputs and outputs for other steps can be published, so everything else can be checked by result verification.

It follows that the way to minimize software verification is to make that critical interval as short and simple as possible: use software to present the ballot, accept selections from voters, and record the votes in anonymous form, then publish the anonymous votes immediately when polls close. The preparation that takes place before the election produces a ballot definition file for the voting machine. If this file is also published, no one needs to verify the ballot preparation software either. Figure 3.6 gives the assurance chart for this case.

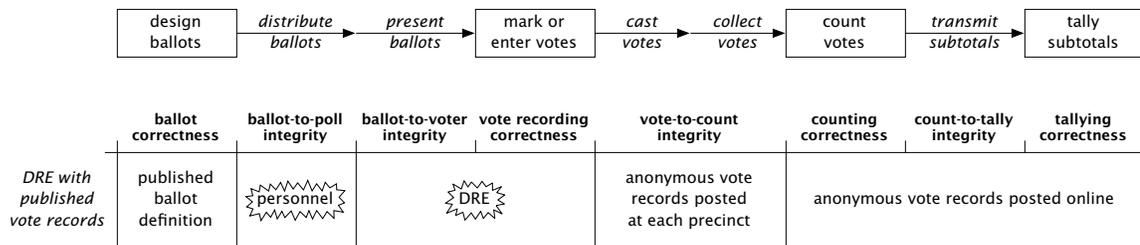


Figure 3.6. Assurance chart for a DRE-based election with published ballot definition and published, anonymous vote records.

In the ballot distribution step, voters have to assume that election personnel have properly distributed the ballot definitions and loaded them into the machines; they have no way to check this for themselves. And in the ballot presentation and vote recording steps, voters still have to trust the software in the DRE machine.

Practical example. Here’s one way that an election with computerized voting but minimal software verification could be carried out in practice.

The software for the voting computer would be written to run on a free computing platform, and finalized and published far in advance of the election so that everyone has time to inspect it and test it. The ballot definition files for the election would be published on government websites, also far enough in advance that members of the public have time to examine them

before the polls open. Anyone would be able to download a ballot definition and run the voting computer software on their own computer to see exactly what will be shown to voters on election day. This provides a chance to detect omitted races, misspelled candidate names, layout errors, and other ballot errors. Thus, the published ballot definition file serves a similar purpose to the paper sample ballot typically mailed to voters before an election.

When a polling place stops accepting new votes at the end of the day, each machine should contain a vote file containing all of its anonymously recorded votes. At this point, every machine would print out a *cryptographic hash* of its vote file; observers can copy down (or photograph) the hashes. A cryptographic hash is a number derived from the contents of a file in such a way that it is easy to calculate the hash for a given file, but difficult to produce a different file that yields the same hash. Publishing the hash makes a public commitment to the contents of the file. (The reason for using a hash is that it is less cumbersome than printing out the entire vote file, but it serves the same purpose.)

The anonymous vote files from every machine would then be published online for all to see after the election. Anyone can calculate the hashes of these files and compare them to the hashes that were printed on election night, to verify that the files are authentic and unaltered. And anyone can count the votes in these files to confirm that the tallying is performed correctly.

The consequence is that neither the ballot layout software nor the vote tallying software would need to be verified. The published ballot definitions, voting computer software, and anonymous vote records would be sufficient to allow members of the public to independently check the accuracy of the election outcome.

What other alternatives for verification are possible?

Electronic ballot markers and printers. An *electronic ballot marker* (EBM) is a computer that marks a paper ballot [80, 81]. The voter inserts a paper ballot and makes selections on the computer, and the EBM prints marks onto the ballot in the appropriate positions. An *electronic ballot printer* (EBP) is a computer that prints out a marked paper ballot. No ballot is inserted; the voter makes selections on the computer, and the EBP prints out a fresh paper ballot that indicates the voter's choices. In both cases, the voter then deposits the paper ballot into a ballot box as usual.

EBMs and EBPs occupy a middle ground between optical scan systems and DRE systems. They provide the flexibility of a computerized user interface for voting, together with a durable paper record that can be recounted later. Like a DRE machine, an EBM or EBP relies on a ballot definition file to describe the choices to present to the voter, and the proper recording of the voter's choices depends on the software running in the EBM or EBP. But the voter now has the option of checking the printed ballot before casting it, instead of having to trust this software. And unlike the printed VVPAT produced by a DRE, this printed ballot is always counted, so the voter's check is more effective.

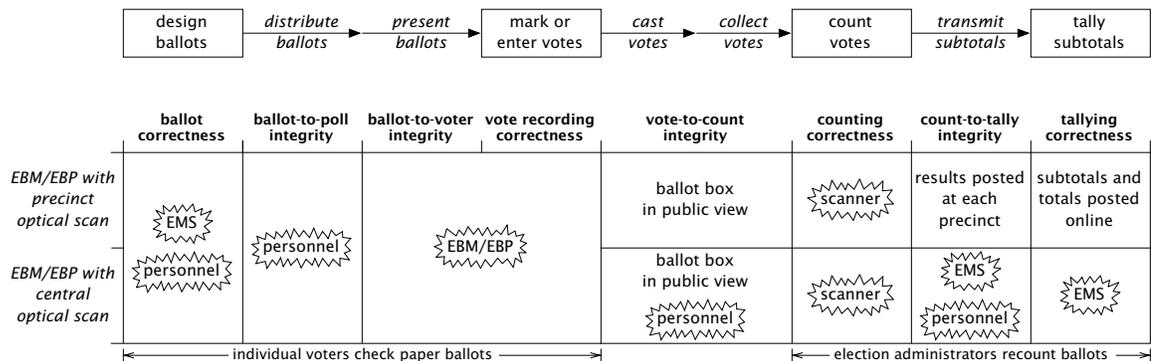


Figure 3.7. Assurance chart for an election with electronically marked or printed, optically scanned ballots.

The corresponding assurance chart, in Figure 3.7, has a left half similar to that of a DRE system, and a right half similar to that of an optical scan system.

End-to-end cryptographic voting. There are several proposed voting systems that provide *end-to-end cryptographic* methods for letting voters verify the election. “End-to-end” refers to the ability of any individual voter to check that his or her ballot survived from one end of the process straight through to the other—from casting to the final result—without special access from election officials.

Recall that earlier in this chapter, I described two possible kinds of publishable records—anonymous vote-revealing records, and identifiable but non-vote-revealing records. End-to-end cryptographic schemes publish records of the second kind as well as the first kind. Examples of these schemes are Punchscan [26], Scratch & Vote [1], Prêt-à-Voter [13], and VoteHere [54]. What they all have in common is that they publish *some* information about each voter’s ballot: enough to let the voter partially check the recorded ballot, but not enough to reveal an actual vote so a voter can sell it. That is, *indirect result verification* is used to ensure the integrity of individual ballots. The partial records are set up in such a way that, with enough voters checking this partial information, the likelihood of an incorrectly posted ballot is nearly zero.

In addition to the partial ballots, actual vote records are separately posted—but these votes have been shuffled so they cannot be associated with particular voters. Anyone can count the posted votes to check the tally. The shuffling is performed using a system called a “mix net,” in which multiple parties participate in the shuffling; no single party learns the total shuffling order, and thus voter privacy is protected.

In these end-to-end cryptographic schemes, the election authorities keep some secret information that enables them to process the ballots into verifiable totals, and the ballots contain serial numbers or cryptographic information as well. In all of these schemes, there is a pre-election audit procedure that lets

voters ensure that this information is consistent and properly formed. After the election, voters can also audit the shuffling procedure to confirm that the posted partial ballots correspond to the posted anonymous vote records, and thus to the tally.

The same mathematical techniques can be applied to votes cast in any fashion (by hand-marked paper, by machine-marked paper, or directly by machine). When hand-marked paper is used, the election can completely escape dependence on software. Figure 3.8 summarizes how assurance is provided in this category of systems.

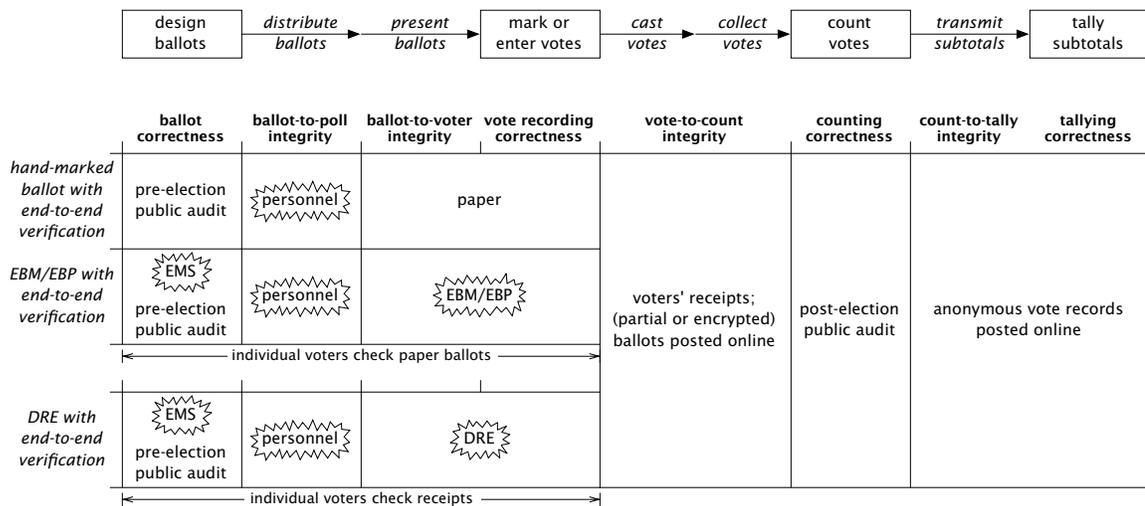


Figure 3.8. Assurance chart for elections with end-to-end cryptographic verification.

Non-cryptographic end-to-end schemes. Of special note are ThreeBallot, VAV, and Twin [67], which provide end-to-end verification without cryptography. These schemes publish all the cast ballots, which anyone can recount to verify the tally. In ThreeBallot and VAV, only *some* of the posted items are identifiable. Each voter's ballot is split into three parts; although all the parts are posted, the voter gets a receipt for only one part—and a single part isn't enough to reveal how they voted. In Twin, each voter gets a receipt for *someone else's* ballot. Thus, while the posted records can be matched with receipts, they can't be identified as belonging to any particular voter. The

assurance chart for all these schemes is similar to Figure 3.8, except there is no need for a post-election cryptographic audit because no encryption or shuffling has taken place.

Comparing voting systems. Figure 3.9 summarizes several types of voting systems on a single chart for comparison.

For conventional paper-based systems, shown at the top, any method of marking ballots (by hand, by EBM, or by EBP) can be combined with any method of counting ballots (by hand count, by precinct optical scan, or by central optical scan). Next come the conventional electronic systems, based on DREs; then the end-to-end cryptographic systems. Finally, at the bottom is the DRE with its ballot definition and results published, as well as a variant of the same scheme using an EBM or EBP instead.

The systems least dependent on software (all other concerns aside) are the hand-marked, hand-counted paper ballots and the hand-marked ballots with cryptographic verification.

If one chooses to exclude the systems with hand-marked ballots (shaded in grey) from consideration, due to the potential usability, accessibility, and accuracy advantages of computer-based vote entry, then the bottom two options in the “public-ballot electronic” category are the least dependent on software. A system based on a DRE with a published ballot definition and published vote records will use the least amount of critical software, but also requires voters to place great trust in that software. A system based on an EBP with a published ballot definition will be dependent on the optical scanner’s software as well as the EBP software, but both software-dependent steps are subject to paper-based checks. The choice between these two options would depend on one’s confidence in the ability to verify DRE software and one’s estimate of the likelihood that significant errors will be caught by observant voters and recounts.

All of the systems that involve entry of votes using any kind of voting computer—DRE, EBM, or EBP—could stand to benefit from easier verification of the software in that computer. This is where we will turn our attention in the next chapter.

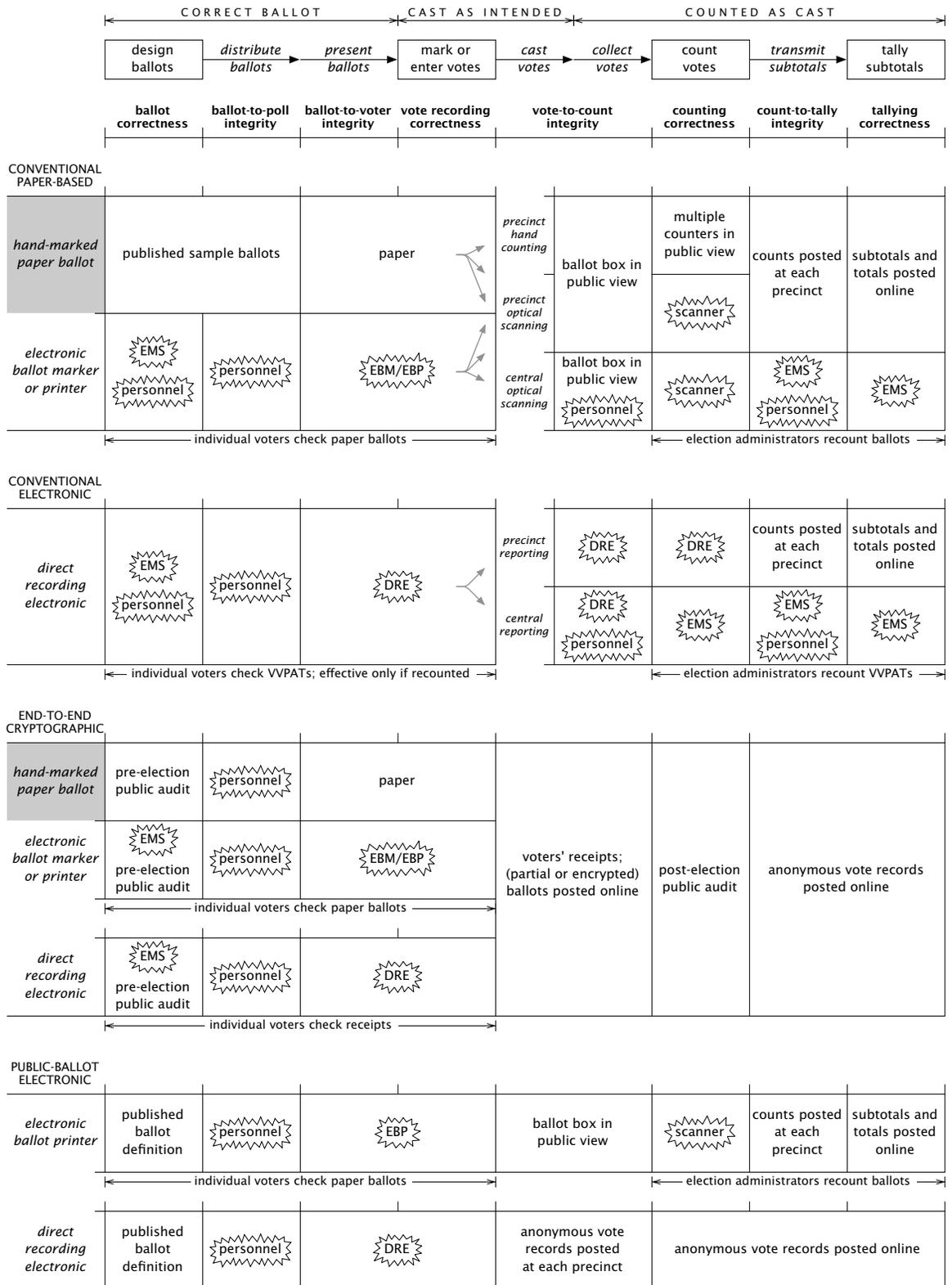


Figure 3.9. Summary of assurance mechanisms for various types of voting systems.

4 Prerendering

How can we make vote-entry software easier to verify?	58
What is prerendering?	59
Why put the entire user interface in the ballot definition?	60
How would a voting computer use a prerendered ballot?	62
What is gained by publishing the ballot definition?	63
What are the advantages of prerendering?	65
How can prerendering be applied to other software?	66
How are votes recorded anonymously?	67

How can we make vote-entry software easier to verify?

For vote-entry software to be easier to verify, we have to make it simpler. The vote-entry software can be simpler if we give it less work to do and shift its responsibilities elsewhere: either earlier, to the preparation stage, or later, to the counting stage.

Shifting responsibilities to the preparation stage is a significant design challenge, but it leads to a dramatic simplification of the vote-entry software. Most of this chapter is devoted to *prerendering*, the technique that makes this possible. Shifting responsibilities to the counting stage means that the vote-entry software should recording votes anonymously with as little processing as possible; this is comparatively straightforward to do and will be discussed in the last section of this chapter.

I developed two prototypes of voting machine software to find out just how small a practical vote-entry program could be. They are called Ptouch and Pvote, described in Chapters 5 and 7 respectively.

What is prerendering?

In a typical voting computer, much of the software code is responsible for generating the user interface for the voter. This includes the code for arranging the layout of elements on the screen, drawing text in a variety of typefaces and languages, drawing buttons, boxes, icons, and so on. In a voting computer with audio features, this also includes code for manipulating or synthesizing sound. (Some voting computers, such as the Avante Vote-Trakker [11], contain speech synthesis software.) The user interface is generated in real time—the visual display and audio are produced (“rendered”) as the voter interacts with the machine.

Prerendering the ballot. The software in the voting computer could be considerably simplified by moving all this rendering work into the preparation stage—*prerendering* the interface before election day.¹ Both Ptouch and Pvote realize this idea.

Today’s DRE machines use a ballot definition that contains only essential data about the ballot: the names of the offices, the names of the candidates running for each office, and so on. But the ballot definition could be expanded to describe the user interface as well. For a visual interface, this would include images of the screen with the layout already performed, buttons already placed, and text already drawn. For an audio interface, this would include prerecorded sound clips. Everything presented to the user would be prepared ahead of time, so that all the software complexity associated with rendering can be taken out of the voting computer.

The ballot definition could specify not just appearance but also behaviour—the locations where images will appear, the transitions from screen to screen, the user actions that will trigger these transitions, and so on. This is exactly the case for both Ptouch and Pvote: the ballot definition is a high-level description of the entire user interface for voting.

¹It was Steve Bellovin who prompted my line of research by suggesting prerendering for voting machines.

Why put the entire user interface in the ballot definition?

Including a complete description of the user interface in the ballot definition, rather than just a set of images, yields several benefits.

Less code in the voting computer. Some of the software in a typical voting computer handles user interface logic. User interface logic tells the computer how to respond to any given user action—for example, to select a candidate when you touch the candidate’s name, or to go to the next page when you press a “Next Page” button. Putting a description of this logic in the ballot definition means the voting computer needs less code for interface logic, just as putting images in the ballot definition means less code for rendering the display.

More thorough public review. If the ballot definition completely describes the user interface, one can review the behaviour of the user interface by examining it. The user interface becomes a separately verifiable artifact.

Compared to the vote-entry software, the ballot definition is more likely to be accessible for inspection by the public, for two reasons. First, there may be fewer legal and political barriers to publishing the ballot definition than the software source code. Second, the ballot definition is a high-level description, which makes it easier to examine than a computer program written in a general-purpose programming language. The result is that more of the voting process is reviewable by non-programmers: both the appearance and the behaviour of the ballot can be inspected without looking at source code for the voting computer.

A more complete public record. The ballot definition file, like any other election information, should be archived and should become part of the public record of the election. It contributes

to making the election a reproducible experiment. In the event of a later investigation, a ballot definition with a complete description of the interface makes it easier to reconstruct the voter experience. Such reconstruction could help investigators evaluate hypotheses about sources of bias or voter error, for example.

Better division of expertise. Separating the user interface definition from the voting machine software mitigates the conflict between accessibility (which requires design flexibility) and security (which requires software simplicity). Instead of playing tug-of-war over the vote-entry software, experts can work independently on what they do best—design can be left to designers, and software security to security experts. Experts in human factors, accessibility, and graphic design can create better ballots themselves, without relying on programmers to implement their designs in code, and without requiring co-operation from voting machine companies. Programmers of the vote-entry software can focus on making the software secure and reliable without affecting the user interface.

Software stability. Regulations that govern ballots can change from election to election and differ from jurisdiction to jurisdiction. Different jurisdictions may prefer to present their ballots differently. Designs will change as we discover better ways to create fair and understandable ballots.

Putting the user interface description in the ballot definition provides the flexibility to handle future changes without having to change the vote-entry software. This means more resources can be devoted toward ensuring that the vote-entry software is correct and secure. It's difficult to complete a rigorous certification process when voting software changes as frequently as it does today, with new versions released every year or two.

How would a voting computer use a prerendered ballot?

In a prerendered-ballot system, the ballot definition is like a small program—a program in an extremely simple language, with limited capabilities. All it can do is present a sequence of images and/or audio clips and accept the user’s selections.

The voting computer simply carries out the program. Thus, the vote-entry software is a *virtual machine* (VM): it abstracts away the details of the computer hardware and its input and output devices. The job of the VM is to respond to user input by displaying images or playing sound clips as prescribed by the ballot definition, keep track of the user’s selections, and record the user’s selections anonymously.

Implementing the VM for a variety of different hardware platforms would enable all of them to use the same formats for ballot definitions and recorded votes—just as other VMs like the Python VM and the Java VM allow a single program to run on different kinds of computers. There can even be multiple implementations of the VM written separately by different people, and as long as they follow a standard ballot definition format, the same ballot definition will work on all of them. For example, there are multiple independently-written Python VMs out there, but most Python programs will run unchanged on all of them.

My hypothesis was that the implementation of the voting VM can be made considerably smaller, simpler, and easier to verify than the software in today’s DRE machines. This dissertation presents Ptouch and Pvote as confirmation of this hypothesis.

What is gained by publishing the ballot definition?

The published ballot definition serves the role of an *electronic sample ballot*, analogous to a sample ballot in a paper election. Standardizing the file format of the ballot definition and implementing the VM for personal computers enables voters to try out the ballot in advance with exactly the same user interface that they will see at the polls. This could be used for training voters as well as testing the ballot.

Verifying the accuracy and fairness of the user interface is critical, because the user interface of any voting machine is in a position to mislead or otherwise influence voters and hence bias the collected votes. The published electronic sample ballot gives the election a *verifiable user interface*, which can be examined by all voters, members of the disabled community, usability experts, and accessibility experts. Anyone could conduct their own user tests of ballots, independent of the voting machine company or the election authority.

Today, less commonly used ballot designs, such as ballots for voters with disabilities or ballots in alternate languages, receive significantly less attention, as only the election office can compose and check electronic ballots. A rather alarming example of this lack of attention occurred at the June 2006 primary election in Santa Clara County, where pollworkers discovered that there was no “continue” button on one of the Chinese screens [40], which made it impossible to cast the Chinese version of the ballot. A published ballot definition would have increased the chances of catching such an error before the election. Publishing an electronic sample ballot helps to level the playing field for members of minority communities and empowers them to play a role in ensuring that the electronic ballot serves them fairly.

Visualizing the ballot definition. Running the ballot definition in a live test might show that the ballot appears to behave

correctly, but it wouldn't be a sure way to test the complete behaviour of the ballot. It would be infeasible to test every possible sequence of inputs. To be certain that the ballot contains no hidden behaviour or incorrect behaviour triggered by rare combinations of inputs, one would have to examine the ballot definition file itself.

In the future, a software tool could be developed to facilitate such examination. The tool would transform an electronic sample ballot into a human-readable format that completely describes the user interface. One possible visualization would be a flowchart-like diagram that illustrates the steps of the user interface with the prerendered screen images. Anyone would be able to download the electronic sample ballot, use the program to produce a diagram, print it out, and examine it. This would make possible a new level of assurance: the electronic voting UI could be verified even by non-programmers. The hardcopy of the UI visualization could also be archived in the records of the election. The visualization alone should be sufficient to reconstruct the interface that voters used at the polls.

What are the advantages of prerendering?

In summary, prerendering the user interface (UI) yields these benefits:

- The *critical software is smaller and simpler*, facilitating its verification.
- The *critical software changes less frequently*, so each release can be tested and audited more thoroughly.
- The user interface can be *designed by designers*, not programmers.
- The *conflict between human factors and security is mitigated*; usability and accessibility can be improved without affecting software security.
- The *conflict between transparency and proprietary interests is mitigated* because less code has to be disclosed in order to evaluate the security of the voting machine.
- The user interface is subject to *broader public review*, since it can be separately published and tested by anyone (not just those who have election equipment).

Standardizing the ballot definition format also yields benefits in interoperability, in addition to the benefits mentioned so far in this chapter.² A standardized format for describing the user interface allows election officials to *mix and match components from different vendors*, leading to increased purchasing power and better product quality, and enabling independently manufactured components to be tested against each other.

²Thanks to David Jefferson for bringing the importance of interoperability to my attention.

How can prerendering be applied to other software?

The prerendering technique can be applied to any kind of software to make verification easier. Applying this technique consists of the following steps:

1. Define a user interface specification language with a set of features that are limited and chosen to suit the intended purpose.
2. Implement a virtual machine that interprets the specification language and presents the user interface it describes.
3. Create user interface designs using the specification language, and publish them for inspection.

Particularly suitable application areas for this technique are those in which a general-purpose computer is used for a specialized purpose, the user interface (UI) is likely to change periodically, and high reliability must be maintained despite changes in the UI. Aside from voting machines, other examples include bank machines, vending machines, and airport check-in kiosks. The user interaction required to operate these kinds of machines is usually limited to a small set of actions, such as selecting from menus and typing in numbers or short pieces of text. This makes it possible to design a simple language for specifying the UI.

In each case, the transaction-handling software can be written once and reviewed thoroughly to ensure its correctness and security. In a voting machine, the transaction-handling software is the part that records the votes; in a bank machine, for instance, this would be the software that communicates transactions to the bank and dispenses cash. The UI can then be easily changed without affecting that critical software—for example, when a bank wants to offer new functionality, a vending machine updates its list of available products, or an airline wants to change the look of its brand.

How are votes recorded anonymously?

In the correctness goals listed in Chapter 2, I identified two components of upholding the secret ballot:

- **Voter privacy:** The processing of voter choices does not expose how any particular voter voted.
- **Coercion prevention:** Voters are not provided any way to present plausible evidence of how they voted to an external party.

Voter privacy. To protect voter privacy, ballots should be stored without any identifying information. The ballots should also be stored in an order independent of the order in which they were cast, so that someone who observes the sequence of voters entering the polling place cannot correlate the sequence of voters with the sequence of stored ballots.

One common method of doing this is to store the vote records in random order, effectively shuffling them as ballots would be shuffled in a real ballot box. Voter privacy depends on the quality of the randomization performed; if the shuffling is predictable, then voter privacy can be compromised. Unfortunately, it's hard to make a computer behave in a truly random way. In fact, independent source code analysis of two leading voting machines (Diebold [12] and Sequoia [7]) discovered flaws in the randomization schemes used for just this purpose. Even worse, randomness is not a quality that can be practically tested, because it is impossible to prove that any behaviour really is random. For example, given a list of numbers, there is simply no way to tell whether the numbers were chosen at random.

A simpler way to avoid revealing the casting order is to sort the vote records according to their contents. (Naor and Teague [49] observed that sorting a list of elements gives them a history-independent representation.) It doesn't matter how the records are placed in order, as long as it's consistent. For example, if there is just one contest with candidates Andrew,

Barbara, and Chris, you could sort the ballots alphabetically by which candidate was selected. Regardless of the casting order, the sorted order will always be the same. Because this method is simple to program and completely obscures the casting order in a verifiable way, this is the method that Ptouch uses.

Coercion prevention. To prevent coercion, voters must not be allowed to put identifying marks on their ballots. In one possible coercion scenario, the coercing party gives each voter a unique secret phrase to enter as a write-in candidate. For example, suppose Ted tells Alice to vote for Carol for President with “moldy explosion” as write-in for Dogcatcher, and also tells Bob to vote for Carol for President with “wrinkled tourbus” as write-in for Dogcatcher. Then the recorded ballots are no longer publishable because they would enable Ted to confirm, and thus buy, Alice’s and Bob’s votes.

One way to resolve this problem is to store each of the voter’s selections as a separate item instead of the entire ballot as a unit. There has been precedent for such a scheme in some paper elections in Switzerland [15], where the ballots are perforated so that they can be separated into strips, one for each contest, before being counted. If an individual voter’s selections cannot be associated with each other, then the voter cannot use a specially marked selection to identify the rest of their ballot.

Storing the ballot in parts might not satisfy election standards that are based around the handling of complete ballot images. For example, the 2005 Voluntary Voting System Guidelines in the United States [80] require DRE machines to “record and retain redundant copies of the original ballot image,” where a ballot image is “an electronic record of all votes cast by the voter, including undervotes” (Section 2.1.2). One way to satisfy this requirement would be to store ballots in both ways: as complete images for non-public auditing, and in separated form for publishing.

5 Ptouch

the touchscreen prototype

Overview	70
Ballot definition format	71
Software design	80
Implementation	83
Evaluation	88
Shortcomings	93

Overview

This chapter describes Ptouch [90], the first prototype vote-entry program I developed, which is designed for a touchscreen voting machine. It provides only a visual interface; the goal was to handle the most common types of elections for fully sighted voters. (Pvote, the second prototype, adds support for most voters with disabilities and for less common types of contests and ballots, at the cost of increased software complexity.)

Ptouch handles contests in which voters can choose one or multiple options (up to a fixed limit) from a list of options, and also allows voters to vote for write-in candidates. This is sufficient to indicate anything that could be expressed by selecting bubbles or arrows on an optically scanned ballot.

The format of the ballot definition forms the core of the design, since it dictates how ballots are designed, displayed, and voted upon. Thus, I'll start by describing the ballot definition format in detail, then proceed to the software itself, which is a VM for displaying ballots in this format.

Ballot definition format

The ballot definition is divided into two parts—the *ballot model* and the *image library*—corresponding to the medium-independent and medium-specific information about the voting user interface (see below). The ballot model specifies the interaction sequence, while the image library specifies the appearance.

Separating the ballot model from the image library reduces the cost and effort of validating changes to the ballot. Replacing the image library is sufficient to adjust the layout or visual style of the ballot, change the display resolution, or translate the interface into another language, all without altering the ballot model. For these kinds of changes, only the new image library needs to be validated, not the entire ballot definition. Comparing two image libraries (for example, to check a language translation) is easier than checking the correctness of a ballot model.

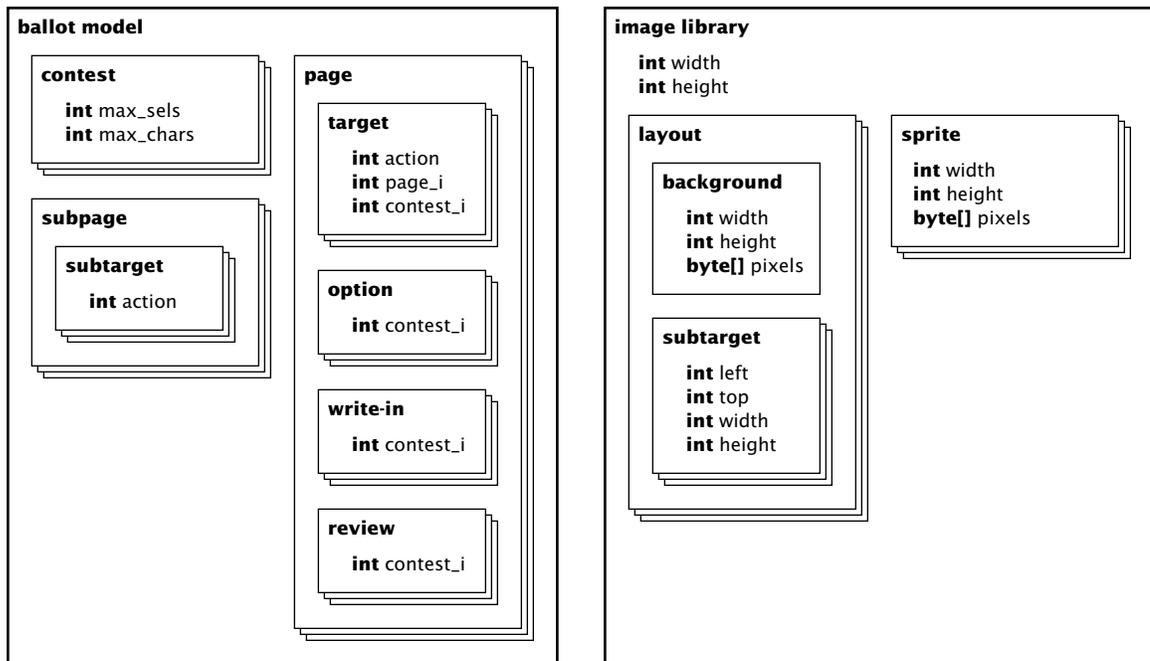
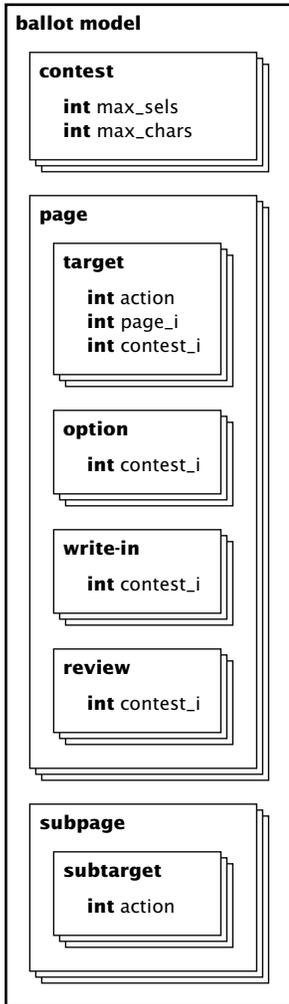


Figure 5.1. The Ptouch ballot definition data structure. Stacked boxes represent arrays.



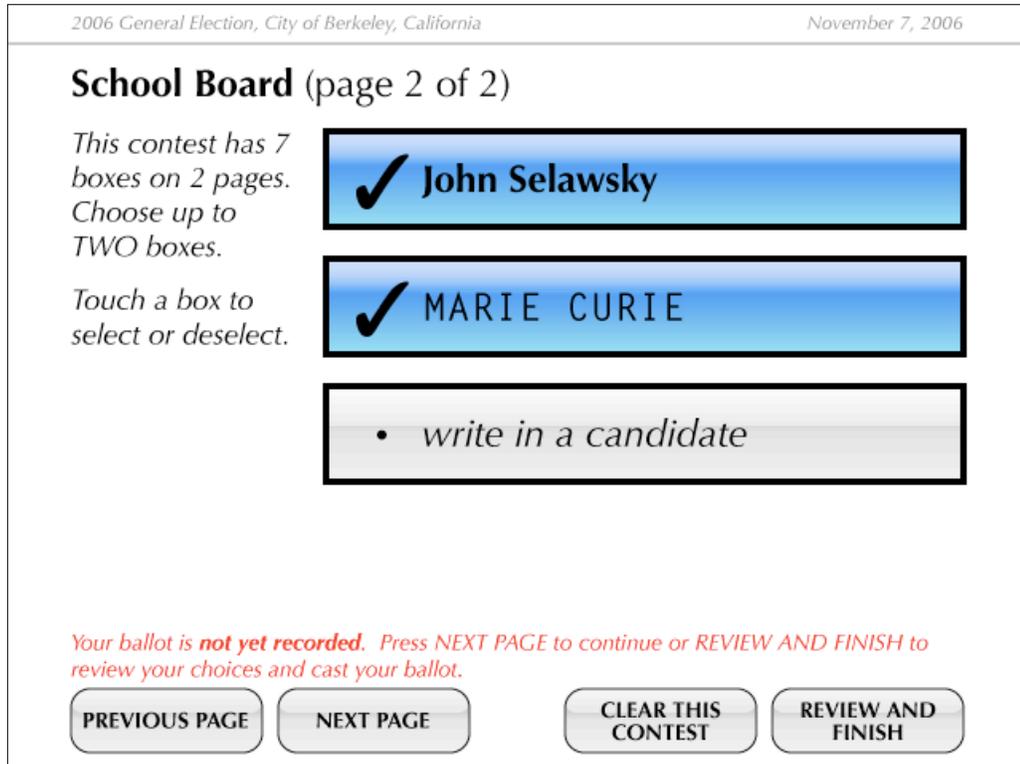
Ballot model. The ballot model consists of an array of *contests*, an array of *pages*, and an array of *subpages*.

A **contest** is a question being put to the voters, such as a referendum on an issue or the election of a candidate (or several candidates) to a position. Each contest has an integer parameter `max_sels` specifying the maximum number of selections that a voter may choose (usually 1, but possibly more in contests that allow choosing multiple candidates) and an integer parameter `max_chars` specifying the maximum number of characters that can be entered for a write-in option.

The **page** is the basic unit of presentation. For example, a single page might display some instructions, a description of a contest, or a list of available options. At any given moment, one of the pages is the *current page*. The user interface begins on the first page in the array of pages. When it transitions to the last page, the ballot is cast with the user’s current selections.

Associated with each page are arrays of *targets*, *options*, *reviews*, and *write-ins*, and any of these can be *activated* by the user. In a touchscreen interface, these elements correspond to rectangular areas of the screen that are activated by touches.

- A **target** is a user-triggered transition to another page. In a touchscreen interface, a target appears as a button that the user can press. Optionally, a target can also trigger one of the following actions:
 - Clear all the selections in a particular contest.
 - Clear all the selections in the entire ballot.
- An **option** is an option that the user can choose in a particular contest. For example, a contest for President would have one option for each of the eligible candidates; a referendum contest would typically have one option for “Yes” and one option for “No.” Each option belongs to exactly one page, though there may be options on different pages that belong to the same contest—for example, if the contest has too many options to fit on one page. Activating an option toggles it between a selected state and an



background image

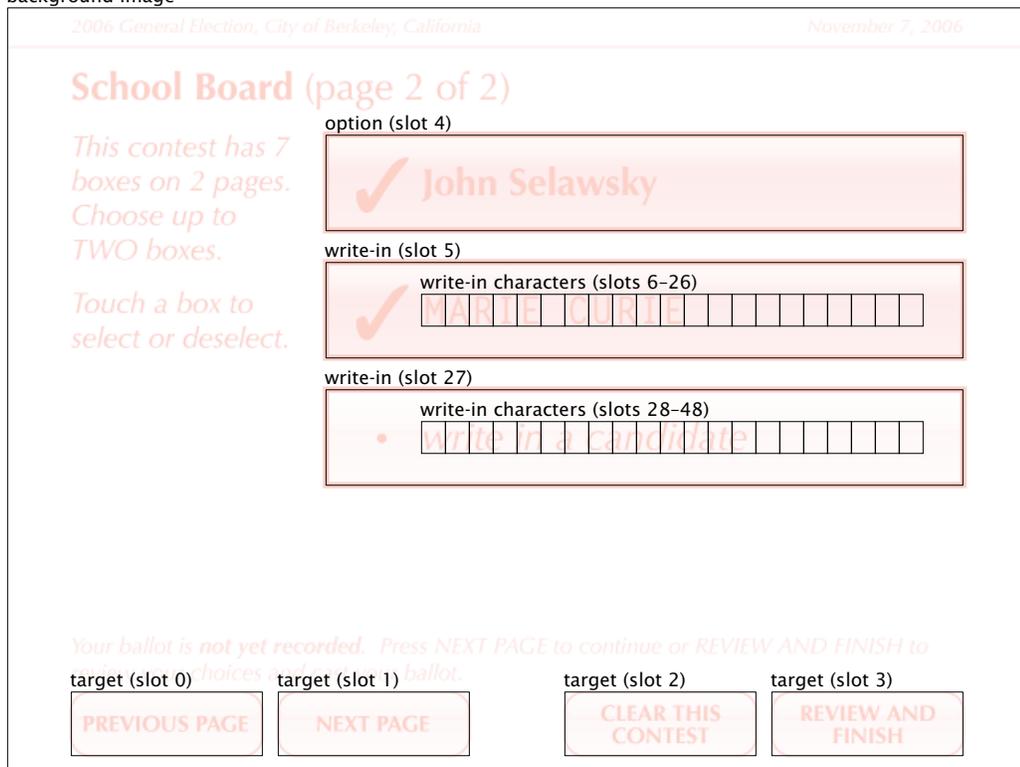


Figure 5.2. A selection page with two options currently selected, and its layout.

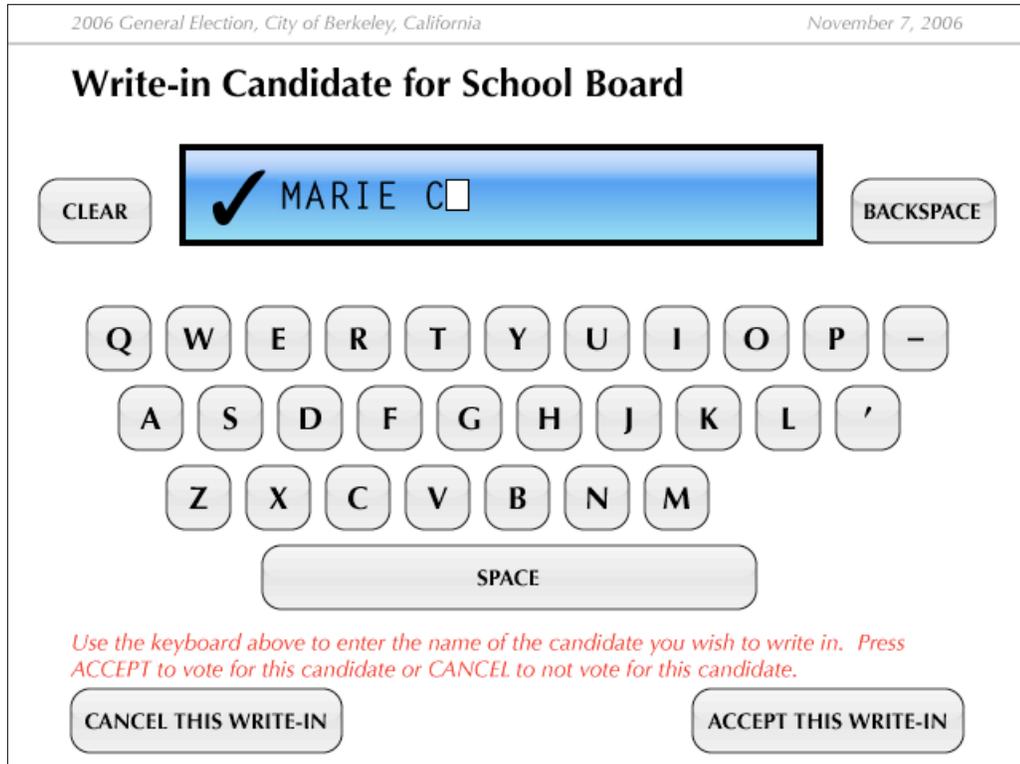
unselected state. In a touchscreen interface, an option appears as a labelled box that changes appearance to show whether it is selected.

- A **write-in** is a write-in option. It can be in a selected or unselected state, just like a regular option; when selected, it also has an associated list of entered characters. When a write-in is activated, it triggers a jump to a *subpage* where the voter can type in the text of the write-in selection.
- A **review** displays the current selections in a particular contest. Activating a review has no effect, though targets can overlap reviews. In a touchscreen interface, a review appears as a screen area (or multiple screen areas) filled in with the option (or options) currently selected in its associated contest. For example, a confirmation page could summarize the voter's selections by presenting reviews for several contests.

A **subpage** is a temporary page for entering a write-in. A subpage is like a subroutine call, but only one level deep—the only possible transition is back to the current page. In a touchscreen interface, a subpage provides a text field and an on-screen keyboard for the voter to type in the name of a write-in candidate. The number of subpages is determined by the contests: there is one subpage for each contest that contains a write-in. A subpage contains an array of *subtargets*.

- A **subtarget** triggers one of these actions:
 - APPEND a particular character to the text field.
 - APPEND2: if the text field is not empty, then append a particular character to the text field.
 - DELETE the last character.
 - CLEAR all the characters.
 - ACCEPT the write-in text and return.
 - CANCEL the write-in text and return.

If the write-in text already contains `max_chars` characters, activating an APPEND or APPEND2 subtarget has no effect. If the write-in text is empty, activating an APPEND2 or ACCEPT subtarget has no effect. If the subpage is exited by an ACCEPT



background image

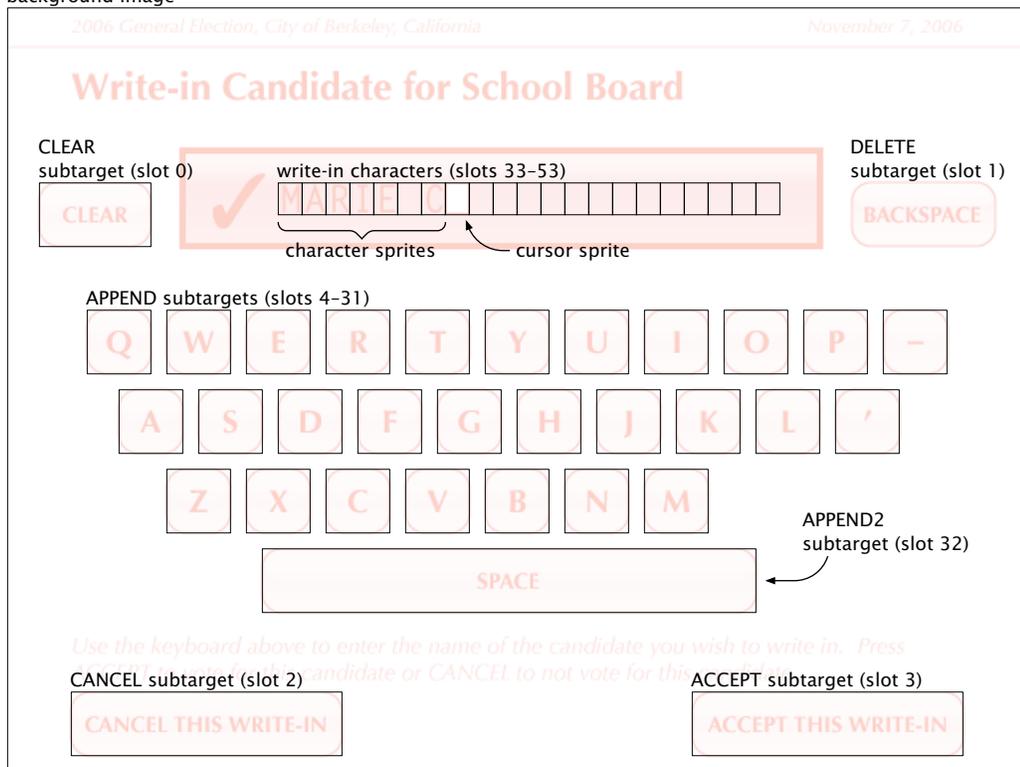


Figure 5.3. A write-in subpage with a few characters entered, and its layout.

subtarget, the write-in option becomes selected and acquires the contents of the text field. If the subpage is exited by a CANCEL subtarget, the write-in option becomes unselected and empty. Thus, it is not possible for a write-in to contain text yet remain unselected.

Because an ACCEPT subtarget only works when there is write-in text present, a write-in cannot be simultaneously empty and selected. The purpose of APPEND2 is to prevent a write-in from *appearing* empty and yet being selected. For example, if the keyboard’s “space” button is an APPEND2 subtarget, then the write-in text cannot consist of only spaces.

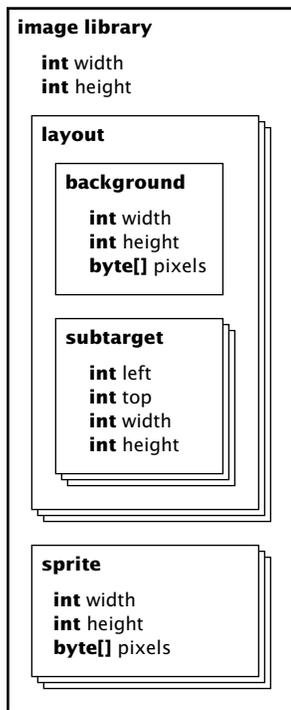


Image library. The image library consists of an array of *layouts* and an array of *sprites*, and also specifies the screen dimensions in pixels.

A **layout** consists of a background image and an array of *slots*. Each page or subpage corresponds to exactly one layout, and vice versa. A **slot** is a rectangular region of the screen where a sprite can be pasted or where a touch will have an effect.

A **sprite** is an image smaller than the screen size that is meant to be pasted into a slot on a background image. The array of sprites contains images of options and write-ins in their selected states, images of characters that for use in a write-in, and the image of the text entry cursor shown while entering a write-in. To keep the DRE software simple, all images are stored uncompressed with 3 bytes per pixel.

In a layout corresponding to a page, the slots correspond to the targets, options, write-ins, and reviews for that page. Each target has one slot, specifying the touch region that activates the target; the image of the target button (or other widget) is part of the background image. Each option has one slot, which specifies both its touch region and also the position for pasting the sprite showing the option in its selected state. The image of the unselected option is part of the background image, and when the option is selected, the sprite is pasted over it. Each write-in also has a sprite for its selected state, which would typically look like a selected option but with space provided for

the write-in text. A write-in has one slot for its touch region and for pasting the selected write-in sprite, and `max_chars` more slots specifying the positions where the entered characters are to be pasted. Each review has `max_sel`s groups of slots (for displaying up to `max_sel`s options selected by the voter). In each group of slots, there is one slot for pasting the selected option sprite and `max_chars` slots for displaying the write-in text if a write-in is selected.

In the layout corresponding to a subpage, the slots correspond to the subtargets and character slots for the page. Each subtarget has one slot, the touch region that activates it. Additionally there are `max_chars` slots specifying the positions where the entered characters are to be pasted.



Referential integrity. To simplify verification, the ballot format minimizes its use of pointers and other kinds of references.

There are only two kinds of references in these data structures:

- Targets refer to the page they transition to. This is necessary to allow for multiple outgoing and incoming transitions to and from each page.
- Targets, options, write-ins, and reviews refer to contests. This is necessary to allow options, write-ins, and reviews to be freely arranged among the pages, so there can be multiple contests on a single page or multiple pages for a single contest.

These references are stored as integer array indices in the ballot definition because it is simpler to verify that an index is in range than to verify that a pointer is valid. All other associations between elements of the ballot definition are implied through structural correspondence. For instance, if there are p pages and q subpages, then there are exactly $p + q$ layouts in the layout array, where the first p are for pages and the last q are for subpages. This use of corresponding array indices avoids the need for pages or layouts to contain pointers to each other.

Similarly, the meanings of the slots are determined by their order in the slot array. The slot array for a page contains, in

order, one slot for each target, then one slot for each option, then $1 + \text{max_chars}$ slots for each write-in, then $\text{max_se1s} \times (1 + \text{max_chars})$ slots for each review. The slot array for a subpage contains one slot for each subtarget followed by max_chars slots for the entered text.

The sprite array contains one sprite for each option and write-in, in the order they appear among the pages, followed by, for each subpage, a character sprite for each APPEND or APPEND2 subtarget and one cursor image sprite.

Well-formedness and validity. There are many possible ways in which one might consider a particular ballot definition to be acceptable; I'll point out two important ones here. I'll use the term *well-formed* to mean that a ballot definition satisfies the assumptions made by the virtual machine implementation. I'll use the term *valid* to mean that a ballot definition represents an acceptable user interface for voting according to the standards of a given jurisdiction.

Because the ballot definition must be well-formed in order for the VM to read it and operate safely and correctly, a verifier in the voting machine checks for well-formedness before accepting a ballot definition. To be well-formed, a ballot definition must meet the following conditions:

- There is at least one page and one contest.
- There is one subpage for each contest that has a write-in.
- There is one layout for each page or subpage.
- Every index referring to a page or contest is in bounds for its respective array.
- Every target or subtarget has a valid **action**.
- Every layout contains the correct number of slots to match its page or subpage, as described in the preceding section.
- All background images match the screen size.
- All slots fit entirely within the screen bounds.
- All option slots, write-in slots, review slots, option sprites, and write-in sprites associated with the same contest have the same size.

- All character slots, character sprites, and cursor sprites associated with the same contest have the same size.
- The image library contains the correct number of sprites to match the ballot model, as described in the preceding section.

Validity, on the other hand, does not have a single definition because it depends on election regulations that can vary by locality. The following are some examples of conditions for validity that are likely to be common, as they prevent some obvious pitfalls and potential sources of confusion in the user interface:

- Target, option, write-in, and review slots do not overlap each other, except that target slots may overlap review slots.
- Character slots do not overlap each other and fit inside their corresponding write-in or review slot.
- Character slots in write-ins and reviews are arranged in the same relative positions as the character slots on the corresponding subpages.
- The user is never trapped in a subgraph of pages, except after arriving on the last page.
- The last page has no target, option, write-in, or review slots.
- There exists some transition path from the first page to every other page.
- Every subpage contains an ACCEPT subtarget, a CANCEL subtarget, and at least one APPEND subtarget.
- Every path that leads to the last page passes through pages that contain reviews for all the contests (thus ensuring that the voter has the opportunity to review all selections before casting the ballot).

Ballot definition files would be produced by ballot design software, such as an interactive tool for laying out and specifying the appearance of a ballot. Such a tool could offer guidance on the usability or accessibility of the design, enforce validity conditions appropriate for a particular jurisdiction, or give notification when validity conditions are not met.

Software design

The Ptouch virtual machine (VM) is composed of four software modules: the *navigator*, the *video driver*, the *event loop*, and the *vote recorder* (see the figure below). This separation does not in itself prevent attacks, as the corruption of any module still has the potential to corrupt the outcome of the election. However, separating the software into modules is a design choice intended to facilitate verification. It is easier to audit and test each module separately when there are limited responsibilities for each module and limited communication between modules.

The **navigator** walks through the pages in the ballot model, always starting on the first page. It keeps track of the current page, the user's current selections, the current subpage (if any), and the entered characters on the current subpage (if any). The navigator responds to just one message:

- When told to **activate** a slot, the navigator takes the action for the corresponding target or subtarget, toggles the corresponding option, or transitions to the subpage for the corresponding write-in.

The navigator issues three kinds of messages to other modules:

- It tells the video driver to **goto** a layout upon transition to a page or subpage. The message specifies the layout index.

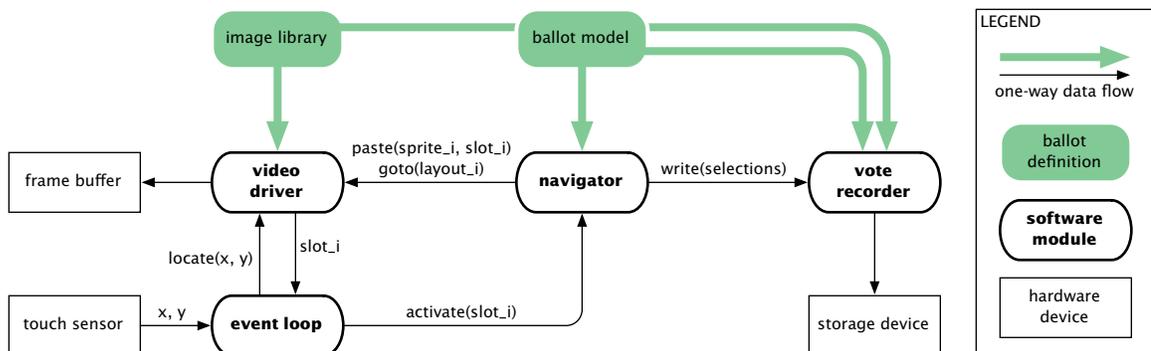


Figure 5.4. Block diagram of the Ptouch VM. The arguments $layout_i$, $sprite_i$, $slot_i$, x , and y are integers; $selections$ is an array of arrays of lists of integers.

- It tells the video driver to **paste** sprites into slots as necessary to display options, write-ins, reviews, and write-in text. The message specifies the sprite index and slot index.
- It tells the vote recorder to **write** the selections when the ballot is cast (when transitioning to the last page). The message contains an array of `max_sel` selections for each contest. Each selection is a list of integers: for a selected option this is a single integer, the index of the selected sprite; for a write-in, this is the index of the selected sprite followed by the indices of the entered character sprites.

The **video driver** has only one piece of state: it keeps track of which layout is the current layout. It interprets the slot index in a **paste** command in the context of the current layout. The video driver handles three kinds of messages:

- When told to **goto** a layout, the video driver copies the background image into the frame buffer and remembers the given layout index.
- When told to **paste** a sprite into a slot, the video driver copies the sprite into the frame buffer at the position specified by the slot.
- When told to **locate** a given point by its co-ordinates, the video driver looks through the slots in the current layout and returns the index of the first slot that contains the point, or a failure code. (When slots overlap, targets take precedence because they come first in the slot array.)

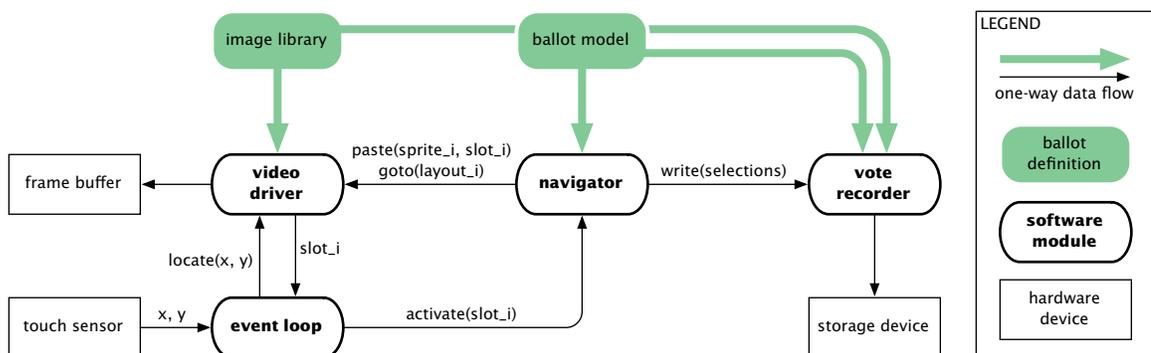


Figure 5.4. Block diagram of the Ptouch VM. The arguments `layout_i`, `sprite_i`, `slot_i`, `x`, and `y` are integers; `selections` is an array of arrays of lists of integers.

The **event loop** receives touch events from the screen's touch sensor. The software assumes that when the user touches the screen, the sensor reports (x, y) coordinates in the same coordinate space used for displaying images. Upon receiving a touch event, the event loop asks the video driver to **locate** the corresponding slot, then passes the slot number on to the navigator in an **activate** message.

The **vote recorder** records the voter's selections in non-volatile storage upon receiving a **write** message from the navigator. The votes are recorded using a tamper-evident, history-independent, subliminal-free storage method. Molnar, Kohno, Sastry, and Wagner have proposed several schemes with these properties [48] for storing ballots on a programmable read-only memory (PROM). Each stored selection includes or indicates its associated ballot definition so that the meaning of the selections is apparent from the storage contents.

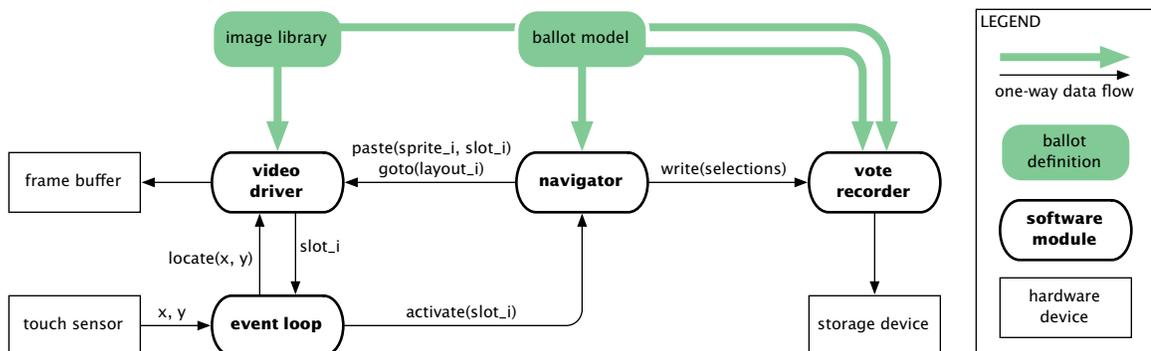


Figure 5.4. Block diagram of the Ptouch VM. The arguments $layout_i$, $sprite_i$, $slot_i$, x , and y are integers; $selections$ is an array of arrays of lists of integers.

Implementation

Ptouch is implemented in Python [63], and it runs on Linux, MacOS, or Windows. Ptouch uses Pygame [62], an open-source multimedia library for Python, to handle graphics and mouse input. It runs on a commodity PC using the video display and the mouse to simulate a touchscreen device (a mouse click at a particular location is interpreted as a screen touch).

Ptouch reads the ballot definition from a file named `ballot` and writes vote records to a file named `votes`. The `ballot` file represents read-only media and is opened read-only; the `votes` file represents a PROM. Each time the program runs, it casts at most one ballot, then enters a terminal state.

Ptouch models the procedures that would take place in a real election as follows. Creating an empty `votes` file corresponds to opening the polls at the beginning of election day with a blank PROM. Restarting the program corresponds to activating the voting machine for a single voter. I have assumed that only the pollworker has the ability to restart the machine, so pollworkers can ensure that each voter only votes once. Setting the `votes` file read-only corresponds to closing the polls and removing the PROM.

The source code for Ptouch is available in Appendix A. The source code is also available online, together with an example of a ballot definition file in the Ptouch ballot format, at <http://pvote.org/>.

Ballot definition. A separate Python module, not shown in Figure 5.4, reads the `ballot` file, verifies all the conditions necessary to determine that it is well-formed, and deserializes it to objects in memory. All integers in the file are stored as 4-byte unsigned integers; images are uncompressed with 3 bytes for each pixel (corresponding to the red, green, and blue components of its colour).

Ptouch does not include any user interface for selecting which ballot definition to use; instead, it assumes that the

appropriate ballot file will be present when the program starts. Different ballot files can be used for different runs.

Note that the selection of a ballot definition can be divided into two parts: choices that have to be authorized by the pollworker (such as choosing which precinct's ballot to use) and choices that the voter is allowed to make (such as choosing a preferred language). The former type of choice can be implemented by having the pollworker select the ballot file. The latter type of choice can be implemented either by having the pollworker select a ballot definition file at the voter's request, or by combining multiple ballots into a single ballot definition. For example, a ballot could support both English and French by including all the pages for an English ballot and all the pages for a French ballot, with a starting page to let the user choose between them.

How the pollworker's selection would be implemented in hardware remains an open question. One possibility would be for the ballot definitions to be stored on individual write-protected memory cards; to support voting for multiple precincts, a pollworker would insert the appropriate precinct's ballot definition card to activate the voting machine for a single voting session. Alternatively, all the ballot definitions could be stored on the machine in advance, and the pollworker would use some other means to choose one when starting each new voting session. In either case, Ptouch models this step simply as having the authorized choice of ballot file be present when the program starts.

Vote storage. The votes file is used to simulate a PROM, a solid-state storage device initially filled with 1 bits; writing to a PROM can change 1 bits to 0 bits, but never the reverse. The vote recorder writes to the file in a manner consistent with this property.

Ptouch stores the ballots using a *copyover list* [48], because it is history-independent, simple to implement, and does not depend on a random number generator. A copyover list is a list of items stored in sorted order; each time items are added to

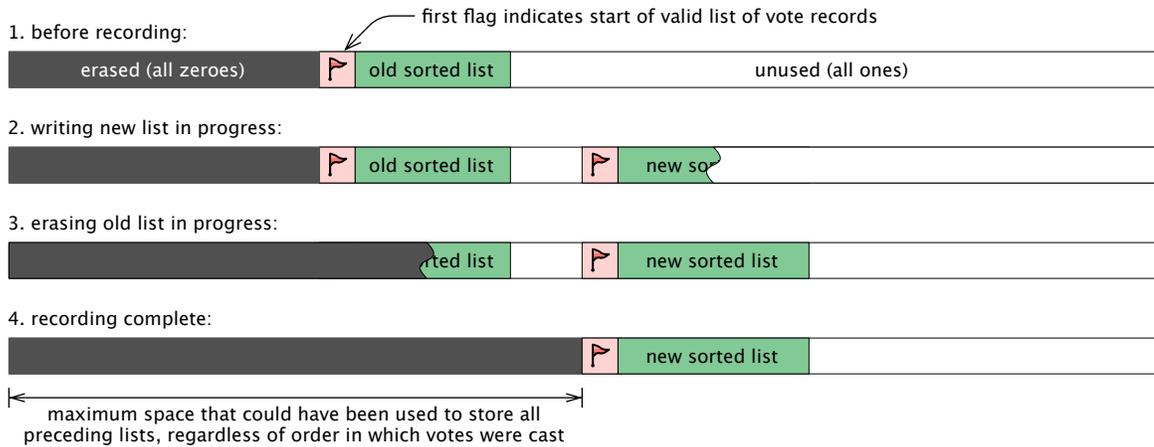


Figure 5.5. Storing votes in a copyover list. The list is always written in sorted order and the amount of erased space preceding the list is independent of the size of previous lists, so that no information is revealed about the order in which votes were cast. On a PROM, changing a bit from 1 to 0 is an irreversible operation.

the list, a new copy of the entire list is written in sorted order and the old copy is erased by overwriting it with zeroes. Because the items are sorted according to their content, the list does not reveal the order in which the items were added. A copyover list uses $O(n^2)$ space in the number of items, but previous analysis [48] shows that only a modest and inexpensive amount of storage would be required to handle all the votes that could be expected to be cast on one machine in one day.

The items in the copyover list are the individual selections within each contest from all the voters. Each item consists of the SHA-1 hash [52] of the ballot definition, the integer index of the contest, and the integer index of the selected option sprite. For a write-in selection, this is followed by the indices of the selected character sprites. All integers are stored as 4-byte unsigned integers. The individual selections are stored as separate items so that the votes file can be published without letting voters mark their ballots to prove how they voted, as explained in Section 4.

Because the items in the list can vary in length, the size of the list depends on the contents of the selections. If the new list

were stored immediately after the old list, the size of the erased space would reveal something about the size of the old list and hence about the sequence of votes. (For example, if two selections are stored, one with a short write-in and one with a long write-in, then the size of the erased space reveals which one was cast first. Casting the long one first would yield a larger erased space than if they were cast in the opposite order.) Therefore, one should always erase the maximum amount of space that would have been required, regardless of the order in which the selections were added to the list.

A flag value is stored at the beginning of each list, and the list is encoded so that it cannot contain the flag value. The first occurrence of the flag in the file is considered to signal the start of the current list of votes. After the new list is written, erasing the flag in front of the old list commits to the new list, as shown in Figure 5.5. This commitment is atomic, because changing even one bit invalidates the flag.

Interpreting recorded votes. For a stored selection to have a well-defined meaning, it must be somehow associated with a ballot definition. Here are four possible ways to do this:

1. Store an entire copy of the ballot definition with each selection.
2. Assume a pre-established global mapping of identifiers to ballot definitions; store an identifier with each selection.
3. Store a cryptographic hash of the ballot definition with each selection.
4. Store an array of ballot definitions, then store an array index with each selection.

The first scheme is simple, but uses a lot of storage space. At a resolution of 1024 by 768 pixels, a full-screen image occupies about 2.4 megabytes; a typical ballot definition is on the order of 10 to 100 megabytes. Storing a few hundred votes would require gigabytes of space.

The second scheme uses very little space, but depends on management of a global namespace of ballot definition identifiers, which might be brittle and error-prone. If a vote

record says that it belongs to ballot definition #34 and there is a disagreement about which ballot definition was #34, the vote record becomes meaningless.

Ptouch uses the third scheme because it uses only a small amount of space, and as long as the hash function is collision-resistant, there can be no ambiguity about which ballot definition is associated with each vote record. As long as you can obtain a copy of the ballot definition, you can ascertain the true meaning of a vote. Since we've already assumed that the ballot definitions are published, this is not a serious problem.

The fourth scheme yields a vote record that is fully self-contained. But in order to store all the definitions on write-once storage, without revealing anything about the order in which they were used, and without using very large amounts of space, all the acceptable ballot definitions must be known in advance. This scheme would make sense for a machine that provides some way for the pollworker to select which ballot definition to use.

If the list of acceptable ballot definitions is fixed in advance, it would be possible to use just one storage device instead of two. The storage medium would initially contain all the ballot definitions; the machine would both read the ballot definitions from it and append the vote records to it. In such an alternative scheme, vote records could not become inadvertently separated from their ballot definitions, but it might be more difficult to provide a hardware-based guarantee that the ballot definitions are never alterable.

Evaluation

Size. The entire implementation of Ptouch is 291 lines long, not including comments and blank lines. The breakdown of module sizes is as follows:

ballot definition loader and verifier	126 lines
event loop	13 lines
navigator	92 lines
video driver	22 lines
<hr/>	
subtotal (user interface)	255 lines
vote recorder	38 lines
<hr/>	
total	291 lines

Dependencies. Ptouch runs on Python version 2.3. It was implemented with minimal dependencies so that the size of the Python code would give a reasonable indication of the true complexity of the program. It uses only one collection type, the Python list. Although some lists change length while the program is running, every list has an upper bound on its length determined by the ballot definition, so an implementation based on arrays could preallocate the necessary space.

The **user interface modules** import nothing from Python's standard library, and use only these built-in functions:

- open and read on the ballot definition file.
- ord to convert characters to integers.
- enumerate and range for iterating over lists.
- len and the remove method on lists.

The only Pygame drawing function that Ptouch uses is `blit`, which copies a bitmap onto the screen. A few other Pygame functions are used just to initialize the graphics display.

The **vote recording module** uses Python's built-in `sha` module for computing the SHA-1 hash of the ballot definition, and also the following built-in functions:

- `open`, `read`, `write`, `seek`, and `tell` on the vote storage file to simulate access to a PROM.
- `ord` and `chr` to convert characters to integers.
- `enumerate` for iterating over lists.
- The `sort` method to sort the copyover list.
- `len` and `max` to find the longest item in the copyover list.

Functionality. The ballot definition format is capable of supporting:

- both general and primary elections
- ballots in any language and any typeface
- voter instructions at any point in the process
- multiple contests on a single screen
- splitting a contest over multiple screens
- contests allowing more than one selection
- photographs or logos shown with candidates
- write-in text in any alphabetic language
- review of selections before casting the ballot
- jumping directly to specific contests or review screens
- regulations requiring voters to review their selections before casting the ballot
- regulations restricting the number of times that voters may review their selections

Because the implementation of write-ins assumes that each character is selected with a single keypress on the touchscreen, it can only support alphabetic languages; write-ins in ideographic writing systems such as Chinese are not supported.

Ptouch does not provide administrative functions such as viewing vote counts or changing configuration settings. It also does not perform encryption; by design, there is no need to encrypt the stored votes.

Separation of concerns. The Ptouch software is divided into five modules that can be implemented and inspected separately. Each module has a limited responsibility, which makes it easier to audit and test.

The ballot definition loader is responsible for establishing that the ballot definition is well-formed. If the loader is implemented correctly, and if the other modules rely only on the conditions of well-formedness, then the only possible kind of software failure is a failure to load the ballot definition. Successful completion of the loading and verification step assures that software errors cannot occur during the voting session.

It is easy to see by direct inspection of the source code that all modules other than the event loop only react to messages they receive. The event loop is the only module capable of initiating messages, but it is also the smallest and easiest to audit.

The video driver is a passive component, never sending any messages at all. In particular, the video driver does not have the authority to activate slots (that is, it cannot “press buttons” in the interface), which reduces vulnerability to errors in its implementation.

The navigator has access to only the ballot model and cannot draw arbitrarily on the display. Because it cannot see the image data, it cannot determine the semantics of the user’s selections. Freezing the implementation of the VM before choosing the order of candidates on the ballot would make it difficult for even the author of the navigator to bias the vote for or against a specific candidate. Also, the only input to the navigator is a slot number, which is a small integer, so the navigator can be subjected to exhaustive testing.

The voting machine has no non-volatile storage other than the ballot definition and the cast vote storage. Because the machine is restarted for each new voting session, and because the ballot definition is read-only, the only state retained between voting sessions is the vote storage. Furthermore, the vote recorder module only receives messages and never sends any messages to any other software module, so no information in the vote storage can reach any of the other modules. Consequently, the user interface seen by each voter is determined only by the ballot definition and cannot reveal any

information about previous voting sessions. Also, this ensures that all voters using the same ballot definition receive the same voting experience.

Election rules. Election regulations concerning the ballot are upheld either by the implementation of the navigator module or by validating the ballot definition.

By design, Ptouch can only cast one ballot each time it runs. It is easy to confirm by inspection of the navigator that the only way to cast a ballot is to arrive at the last page and to see that the last page is a terminal node in the ballot definition.

It is also straightforward to verify that overvoting is impossible, because only the navigator can manipulate the user's selections, and there are only two places in the code where an item is added to the selection list.

Other election process rules can be verified by examining the ballot definition. For example, to ensure that the voter will be notified of undervotes before casting the ballot, we would check the graph of transitions among pages to see that the voter must proceed through review pages before arriving at any page that can cast the ballot.

Comparison. At only 291 lines of Python, the Ptouch code is much smaller than the 31 000 lines of code in Diebold's AccuVote TS software.¹ It may be slightly more appropriate to compare the 255 lines of UI code with the AccuVote's 14 000 lines of UI code—but neither comparison is entirely fair, because Ptouch lacks some of the AccuVote's functionality and the two systems have different sets of dependencies. Nonetheless, the correctness of Ptouch is certainly easier to assure than the correctness of the AccuVote TS code. In general, programs with less code tend to be easier to review, easier to test, less likely to contain bugs, and less likely to crash.

One reason that there is less code is the choice of programming language: Ptouch requires a Python interpreter, whereas the AccuVote TS does not. On the other hand, the

¹This is less than Kohno's figure of 49 609 lines [43] because it excludes blank lines and comments.

AccuVote TS software depends on Microsoft Windows CE and builds its user interface using the Microsoft Foundation Classes, which are much larger and more complex than the `blit` functionality that Ptouch uses from Pygame.

It is not unreasonable to consider running Python on voting machines. Python is widely deployed and vetted and is supported by an active developer community. Unlike Windows CE and MFC, Python is a mature open source project, distributed with an extensive suite of regression tests. As a data point concerning Python's size, note that Nokia has released a small Python interpreter that runs on Nokia mobile phones [57]. The interpreter fits in a 504-kilobyte installation package, which also includes over 40 Python library modules that Ptouch doesn't use.

Alternatively, the Python code could be translated into a compiled language. Although Ptouch is written in a higher-level language, it uses very few of Python's library modules and built-in functions, as described earlier in this section. It is reasonable to expect that translating this code into a compiled language would multiply its size by a factor of 3 or 4, but not by 100.

Despite its small size, the Ptouch code maintains clear boundaries and minimal data flow among its five modules. As described earlier in this section, many of the desired security properties of the voting machine are straightforward to verify in Ptouch, due to its design. The AccuVote TS code does not lend itself to similarly easy analysis.

Shortcomings

Ptouch lacks several kinds of important functionality.

Accessibility. Ptouch only supports a touchscreen for both input (receiving choices made by the voter) and output (displaying information to the voter). Thus, it is not usable by voters who are blind or voters who lack the motor control to accurately touch buttons on the touchscreen.

Printing. Ptouch does not accommodate a printer, so it does not produce any permanent paper records. In particular, there is no voter-verifiable printed record of votes (VVPAT), a feature that is currently required by law (either for elections or for purchase of new equipment) in 16 U. S. states [23]. As of this writing, the United States Congress is considering a bill [79] that would make VVPATs a nationally required feature on all DRE machines.

Audit logging. Ptouch does not record any logs of its operation. Audit logs can be of invaluable assistance to investigations in the event of a dispute, evidence of tampering, or a software error.

Straight-party voting. Some paper ballots offer a way to make a single party selection that has the effect of voting for the candidate of that party in every contest. As of this writing, straight-party voting is used in 17 U. S. states [50], but Ptouch does not support such a feature.

Complex voting rules. Some ballots have voting rules that cross between selections or contests. For example, sometimes primary elections for multiple parties are combined on a single paper ballot, where the voter first indicates their choice of party and then votes in the contests for that party's primary. Ptouch would not be able to present different contests depending on the party selection that the voter made. As another example, a

ballot for a recall election would first let voters vote for or against the recall itself, then offer a selection of replacement candidates. Typically, it is only valid to vote for a replacement if one has voted in favour of the recall. Ptouch cannot enforce this kind of restriction.

Ranked and other election methods. Most single-winner elections decide the victor by the *plurality* rule (also known as “first past the post”), in which each voter votes for a single candidate and the candidate with the most votes wins. Despite its popularity, it is a poor method for electing a single winner because it penalizes moderate candidates and often motivates voters to misrepresent their preferences [44], locking in polarized two-party control of the government. Of the many election methods that have analyzed by social choice theorists, it is one of the worst methods for electing a single winner.

One simple way to obtain a truer representation of voter preferences is *approval voting* [9], in which each voter can vote for as many candidates as they want. An approval election is easily conducted with Ptouch by setting `max_sel`s equal to the number of candidates.

Another election method that has been proposed is *range voting* [74], in which voters assign scores to the candidates and the candidate with the highest average or total score wins. Range voting can be conducted with Ptouch by setting up a ballot with a separate contest for each candidate. For example, to allow scores from 0 to 10, the ballot can simply present eleven choices, numbered 0 to 10, next to each candidate.

Several election methods involve ballots on which voters can rank the candidates. The Schulze method [71] and the Tideman method [77] belong to a family of methods called Condorcet methods, which use ranked ballots to simulate all the possible one-on-one match-ups among the candidates. With these methods, voters are allowed to specify rankings that include ties (i. e., they can assign the same rank to more than one candidate). Another notable method, in which voters must specify rankings without ties, simulates a series of runoff

elections in which the least popular candidates are successively eliminated. This method is known as “preferential voting” or the “alternative vote” in many countries around the world and called “instant runoff” in the United States.

Ptouch does not provide a way for voters to rank their choices. Also, because Pvote records each selection separately, multiple selections cannot be combined to produce the effect of a ranked ballot.

For example, some paper ballots implement ranking by repeating the same list of options multiple times. San Francisco uses a simplified variant of “instant runoff” in which voters rank only their top three choices. On the ballot, the same list of candidates appears in each of three columns; voters are instructed to indicate their first choice in the first column, second choice in the second column, and third choice in the third column. This tactic would not work for Pvote because Pvote would store the voter’s first, second, and third choices as three separate selections, dissociated and scattered among all the selections made by other voters.

6 Accessibility

Why was a second prototype needed?	97
What is Pvote's approach to accessibility?	98
How are alternative input devices handled?	99
How does blindness affect interface navigation?	100
How do blind users stay oriented within an interface?	101
How do blind users keep track of what is selected?	102
How do blind users get feedback on their actions?	103
How are vision-impaired users accommodated?	104

Why was a second prototype needed?

Ptouch, the first prototype, demonstrates significant progress in simplifying voting machine software, but it lacks several key abilities, as explained at the end of the last chapter. It cannot handle certain ballot features, it does not print a paper record, and—most significantly—it supports only a touchscreen for input and output. Such an interface can only be used conveniently by voters who can see, who can read, and who have sufficient fine motor ability to accurately select items on the screen.

A major motivator for using electronic voting machines in the first place is to meet the accessibility requirements dictated by HAVA [78]. By failing to support more accessible voting interfaces, Ptouch left open the question of just how much software complexity is necessary to fulfill these machines' ostensible reason for existing. The purpose of Pvote, the second prototype, is to answer that question, and to show that better verifiability can be achieved without sacrificing accessibility and useful functionality.

What is Pvote’s approach to accessibility?

When I began working on accessibility support, I started to create a special “accessible version” of the system just for blind users, with a keypad for input, an audio-only interface, and no visual display. Before long, however, it became apparent that a *universal design* approach would be more fruitful.

Universal design [75] is the practice of designing artifacts that are flexible enough to support a wide range of users with and without disabilities, instead of separate artifacts or assistive devices for specific disabilities. A unified solution avoids stigmatizing people with disabilities, and the increased flexibility often yields benefits for all users. Volume controls on public telephones are an example of universal design: they help everyone use the telephone more easily in a noisy environment, not just those who are hard of hearing.

Pvote’s unified solution is a single user interface with *synchronized audio and video*, rather than a visual interface for sighted voters and a separate audio-only interface for blind voters. The same information is presented concurrently in audio and video; user input always yields both audio and visual feedback. Voters without disabilities can also benefit from audio confirmation of their choices [73].

Noel Runyan, an expert on accessible technologies, recommended synchronized audio and video to me during the early stages of this work. His recent report on voting interfaces [69] also makes this recommendation. Although not all of the electronic voting machines currently in use support synchronized audio and video, such a requirement is present both in the 2005 Voluntary Voting System Guidelines (VVSG) [80] (item 3.2.2.1f) and in a draft of the next generation of these guidelines [81] (item 3.3.2-D).

How are alternative input devices handled?

Pvote takes a universal design approach to input devices as well as output devices. Its design is intended to support voting hardware with both a touchscreen for input and an alternate input device. The design assumes that the alternate input device consists of a fixed number of momentary buttons and sends a signal identifying a button whenever a button is pressed. This is a useful input model because it allows a wide range of devices to serve as the alternate device, including a regular keyboard, a numeric keypad, a set of hardware buttons designed for voting, or a sip-and-puff device. The voter can decide whether to use the touchscreen or the alternate input device, and can mix them freely.

This simple input model does not account for the timing of button presses. For a person with severe physical disabilities who can only operate one or two buttons, the length and timing of button presses is an important way to convey information. Although Pvote cannot distinguish between a short press and a long press, these inputs could be translated in hardware to separate signals. That is, from Pvote's perspective there would be two different buttons: the hardware would send one keycode for a short press and a different keycode for a long press.

However, Pvote's input model does not support *autoscan*, a typical feature of "single-switch access" software. In an autoscan interface, a cursor cycles through a list of choices at a steady rate and the user activates the switch when the cursor arrives at the desired choice.

A system with both multimodal input and multimodal output is helpful not only for blind voters but also voters with low vision, voters who are illiterate, voters with cognitive disabilities, and voters with physical impairments that make it hard to use a touchscreen, as well as voters with multiple disabilities.

How does blindness affect interface navigation?

With respect to voting user interfaces, the visual channel has two advantages over audio. First, it can convey textual information at a higher bandwidth: for most people, reading a printed list of candidates' names is faster than listening to them spoken aloud. Second, a visual image can convey more information at once without an explicit navigation mechanism: although a screen full of text probably exceeds what a person can hold in working memory, a sighted person can easily select and gather information of interest just by looking around at different parts of the screen.

A consequence of both of these properties is that audio-only voting interfaces require smaller units of navigation than video-only voting interfaces. Whereas an entire page can be visually “current” to the voter, only a few words can be aurally “current” at any given moment. For example, a visual interface can present an entire list of candidates at once but an audio interface must present the candidates one at a time. Therefore, a multimodal interface should support the notion of the user's focus at two different levels of hierarchy, with audio information at the finer-grained level. Pvote introduces *states* within pages to serve this purpose.

How do blind users stay oriented within an interface?

Visual information can be presented passively, whereas presenting audio information requires continuous activity. Even an inert display can convey visual information, whereas silence conveys no audio information at all.

If a user is distracted while viewing static visual information, then getting reoriented is just a matter of looking over the information again. But if a user is distracted while listening to audio, then getting reoriented requires that the computer actively replay the audio. Therefore, an audio interface needs fallback mechanisms to trigger reorientation. The ballot definition needs to be able to specify a “Where am I?” button that the user can press to recover context.

There also needs to be a way to provide reorienting information after a period of inactivity, if the user is lost and doesn’t know what button to press for help. The Pvote ballot format has a timeout parameter for this purpose (see Figure 7.2); the ballot definition can specify a transition to another page or audio message to be played when the timeout period expires with no user activity. The most recent draft of the next version of the VVSG [81] includes requirements for a “defined and documented inactivity time” (item 3.2.6.1-E) after which the system alerts the user (item 3.2.6.1-F); Pvote’s timeout functionality addresses these requirements.

How do blind users keep track of what is selected?

At any given moment, the voting machine keeps track of the set of current selections in each contest, which I'll call the *selection state*. Recall that in Ptouch, the selection state is displayed visually by *option areas*, which display a particular option, with one appearance if it is selected and another if it is not, and by *review areas*, which list all the selected options in a specified contest.

To communicate the selection state to a blind user, the audio interface needs to be able to play audio messages that vary depending on what is currently selected. Thus, a Pvote ballot defines audio in terms of a sequence of *audio segments*, where each segment can be constant or variable. A *constant segment* always plays the same audio clip independent of the selection state; a *variable segment* selects an audio clip to play as a function of the current selection state. Constant and variable segments are concatenated together to give the effect of filling in blanks in spoken prose, yielding a verbal description of the selection state.

How do blind users get feedback on their actions?

Not every user action succeeds. For example, the user should not be allowed to overvote. Ptouch enforced this rule, but provided no particular feedback; an attempt to select an additional candidate would simply have no effect when the contest is already full. (I use “full” to mean that the maximum allowed number of selections in the contest is selected, and “empty” to mean that none of the options in the contest are selected.)

In a visual interface this might be considered acceptable behaviour, as the user can immediately see whether or not the attempt to select had an effect: either the candidate’s name takes on a selected appearance, or it doesn’t. But in an audio interface, there is no such direct feedback without an audio message describing what just happened. Therefore, to support audio-only voters, the ballot definition needs to be able to specify different audio messages depending on whether an action succeeded or failed, and possibly also depending on the reason for success or failure. The new *condition* structure in Pvote’s ballot format makes this possible (see Figure 7.2).

How are vision-impaired users accommodated?

A large-type mode and a high-contrast mode can be helpful for users with a vision impairment. Both the 2005 VVSG [80] (items 3.3.2.1b and 3.3.2.1c) and the draft new guidelines [81] (items 3.2.5-E and 3.2.5-I) require electronic voting displays to be capable of showing all information in at least two type sizes, 3.0–4.0 mm and 6.3–9.0 mm, and to have a high-contrast mode with a contrast ratio of at least 6:1 (on current voting machines this usually means a black-and-white mode).

Ptouch can already accommodate these requirements by providing multiple prerendered versions of the ballot in a single ballot definition file, together with buttons for selecting or switching the desired presentation mode. For example, each normal-type page could include a button for switching to the large-type version of the same page. However, such a ballot would contain duplicates of the contests and their options. In terms of the ballot definition data structures, the large-type contest and the normal-type contest for each office would be distinct contests with distinct options. Ptouch's electronic records of votes would therefore reveal whether the voter selected a large-type candidate or a normal-type candidate, which could be considered a voter privacy violation.

Because Pvote has more flexible handling of user input, it is possible to design ballots for Pvote that avoid this problem. A single user action can trigger multiple effects in Pvote, so user selection of any one option can be made to automatically select all the corresponding variants in the other display modes (e.g., touching the button for Jane Smith in normal print also selects Jane Smith in large print, Jane Smith in high contrast, etc.). The results of making the same selections in different presentation modes would then be indistinguishable.

7 Pvote

the multimodal prototype

Overview	106
Goals	107
Design principles	110
Differences between Pvote and Ptouch	114
Ballot definition format	121
Software design	127
Implementation	132
Evaluation	133

Overview

This chapter describes Pvote [91], the second prototype vote-entry program I developed. Unlike Ptouch, Pvote offers support for most voters with disabilities by providing synchronized audio and video output, and also by accepting input from buttons and other accessible input devices as well as touchscreen input. In addition, Pvote handles several less common ballot features that Ptouch does not support.

Pvote is intended for voting machines that are electronic ballot printers; thus, both the ballot definition and the VM software contain a component specifically to support ballot printing. An implementation targeted for other types of voting machines could substitute a different component for recording the cast votes, such as the tamper-evident direct recording mechanism in Ptouch.

Goals

Pvote aims to achieve both functionality goals and security goals. The set of supported ballot features and user interface features is determined by the ballot definition format. Security depends on the correct and verifiable implementation of the Pvote program.

Functionality. Voting systems should be highly usable by voters of all kinds, and their usability should be evaluated and improved through user testing. However, user testing of specific ballot designs is outside the scope of the present work. The aim here is to design not a particular ballot, or even a particular style of ballot, but a ballot definition format—one flexible enough that usability and accessibility experts can use it to create better and better ballots as our understanding of voting human factors improves. As explained in Chapter 4, the prerendering approach opens up the process so ballot design can be done by expert ballot designers, not just voting machine programmers.

If the ballot definition format is rich enough to replicate what existing voting machines do, then the resulting voting system will be capable of being at least as usable as today's voting systems. We can be assured of not having lost ground in usability, while throwing open the door to future ballot designs with better usability. Thus, the goals for the new ballot definition format are described in terms of sufficient functionality to match existing systems:

- It should be possible, with an appropriate ballot definition and corresponding hardware, to produce a similar or better user experience compared to existing electronic voting systems, including those that support audio or synchronized audio and video.
- It should be possible to define a reasonably usable synchronized audio and video interface corresponding to a real ballot.

- It should be possible to create a single ballot definition that makes sense for a voter who can only hear the audio and also makes sense for a voter who can only see the visual display.
- It should be possible to implement most of the voting features needed for real elections, such as multiple-selection contests, write-ins, straight-party voting, eligibility for contests dependent on selections in other contests, restrictions on cross-endorsed candidates, and ranked voting.

Security. As elaborated in Chapter 2, the essential task of a voting system is to obtain an accurate and fair measurement of the preferences of the electorate. Pvote aims to uphold the security goals given on page 31 of that chapter:

- G3. In every voting session, the correct choice of ballot style is presented to the voter.
- G4. Every ballot is presented to the voter as the ballot designer intended.
- G5. At the start of every voting session, no choices are selected.
- G6. The voter's selections change only in accordance with the voter's intentions.
- G7. The voter receives accurate feedback about which choices are selected.
- G8. The voter can achieve any combination of selections that is allowable to cast, and no others.
- G9. The voter has adequate opportunity to review the ballot and make changes before casting it.
- G10. The ballot is cast when and only when the voter intends to cast it.
- G11. Every selection recorded on a ballot cast by a voter is counted.
- G12. No extra ballots or selections are added to the count.
- G13. The selections on the ballots are not altered between the time they are cast and the time they are counted.
- G14. The tally is a correct count of the voters' selections.

- G17. No voting session allows more than one ballot to be cast.
- G20. Every voter can begin a voting session within a reasonable, non-discriminatory waiting time.
- G21. Every voting session provides a reasonable, non-discriminatory opportunity to cast a ballot.
- G23. The processing of voter choices does not expose how any particular voter voted.
- G24. Voters are not provided any way to give plausible evidence of how they voted to an external party.

With Pvote:

- G3 has to be upheld by the pollworker who selects the ballot style for the voter.
- G4, G5, G6, G7, G8, G9, and G10 are upheld by verifying that the ballot definition is properly designed and by verifying that Pvote interprets the ballot definition correctly.
- G11, G12, and G13 are upheld by the physical procedures for casting and handling the paper ballots printed by Pvote.
- G14 is upheld by the counting procedures for paper ballots.
- G17 is upheld by verifying that Pvote becomes inert immediately after casting a ballot.
- G20 and G21 are upheld by verifying that Pvote does not crash or become unresponsive during a voting session.
- G23 is upheld by ensuring that Pvote's behaviour in each voting session is independent of all previous sessions.

The security goal is that it must be possible (and preferably easy) for reviewers to verify to their satisfaction that the system guarantees the necessary correctness properties, without relying on faith in the honesty or competence of the system's developers.

Design principles

In the design for Pvote's ballot definition format, I tried to anticipate and support many kinds of functionality. Because the design involved many trade-offs among interdependent factors, I found that I had to choose some guiding principles to help keep design decisions well grounded. These principles would probably also be useful when taking the prerendering approach to high assurance in other domains as well as voting. The next few sections outline these principles, in order of decreasing priority.

Work from a concrete use case. I found it helpful to examine a specific paper ballot (in this case, a sample ballot from the November 2006 election—Contra Costa County's ballot style 167) and consider what would constitute an acceptable corresponding electronic ballot. Any faithful translation of this ballot into electronic form must present all of the information on the paper ballot, enable a voter to navigate through the ballot, keep the voter oriented as to their position in the ballot, allow access to all available options, and keep the voter aware of the current state of their selections. The electronic ballot must achieve all of these things for voters using only the visual display as well as voters using only the audio.

The paper ballot turned out to be invaluable for driving the design process. It was often a good idea to refer back to the paper ballot to work out exactly what should appear on the screen, what audio should be played, and the appropriate responses to all possible user inputs. The exercise of creating a specific ballot definition file revealed which features had to be supported by the ballot definition language and when it was necessary to add more capabilities to the VM.

Minimize VM complexity. The ultimate goal of this work is to facilitate the review of the software that has to be verified—in this case, the VM. In general, the smaller and simpler the VM,

the easier it is to verify. When faced with a design decision, I would keep returning to this goal and choose whichever option yielded a smaller or simpler VM. This principle was secondary only to including the necessary functionality to implement a real ballot, as described in the preceding section.

One consequence of this principle is that it is more important to avoid redundancy in the VM code than to avoid redundancy in the ballot data. For example, although the ballot definition file is likely to contain images that are highly compressible, they are not compressed, because that would require additional decompression code in the VM. Security reviews are expensive, but storage is cheap.

Maximize UI design flexibility. Other things being equal, it is better for the ballot definition language to allow a wider range of user interfaces to be specified. Giving more expressive power to the ballot definition makes the VM less likely to have to change to support new user interface designs. Since each change invalidates previous software reviews, future-proofing the VM yields real security benefits. Thus, when considering design options that do not significantly differ in the complexity of the VM or in the ability of the VM to enforce correctness constraints, the preferred option is the one that leads to a larger space of possible user interfaces.

One effective way to make the ballot definition language more expressive is to embrace orthogonality in language primitives. Replacing specialized high-level constructs with a combination of more general-purpose primitives can be doubly beneficial: the increased generality enables more possibilities to be expressed, while the increased uniformity makes the implementation in the VM more concise. For example, the new ballot definition language has no special cases to distinguish, say, review screens or write-in screens from other kinds of screens; all of these are just pages, and information can be freely arranged on each page.

The trade-off is that using lower-level constructs sometimes makes the ballot definition more tedious to review. Switching to

more general, lower-level constructs tends to be advantageous if it gives the UI designer more flexibility without creating new ways of violating correctness, and if the additional tediousness of reviewing ballot definitions can be mitigated by automated tools for reviewers.

Maximize UI review efficiency. In the prerendering paradigm, assurance is derived from human review of the user interface specification (which, in this application, is the ballot definition). It's impossible to eliminate the necessity of human involvement in evaluating the correctness of the user interface—whether a visual display or a spoken message is misleading is a judgement that can only be made by a human reviewer.

However, design choices in the UI specification language can affect the level of confidence with which a human reviewer's observations can be generalized across all of the situations a user might encounter in using the voting interface. A well-designed ballot definition language can give human reviewers the leverage to draw broad conclusions from manageable amounts of review and testing.

In any system with even a modest number of variables, the number of states that the system can be in is likely to be so large that a human reviewer cannot observe the user interface in every possible state. But the ballot definition language can defend the human reviewer from this combinatorial explosion of states. The language can facilitate the creation of ballot definitions for which observing a limited number of states (for example, walking through the ballot making selections as in typical pre-election testing) is sufficient for a reviewer to accurately extrapolate the UI presentation of all the states the system could come to be in.

For example, candidate's names are spoken in the audio interface in several contexts. When the voter selects Candidate X, there should be an audio confirmation message such as "Candidate X has been selected." When the voter is reviewing selections, the voter should hear a message such as "For President, your current selection is Candidate X."

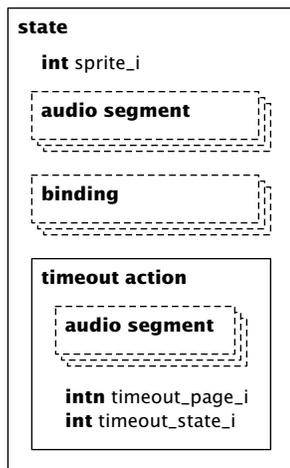
Suppose that these two messages were each independently recorded as a single sound clip. In order to verify the correctness of the audio, a human reviewer would have to listen to each pair of messages to ensure that the candidate sounds the same in each pair—it would not do for the selection message to say “Candidate X” but for the review message to say “Candidate Y.” In such a scheme, the number of messages to review would be roughly the number of candidates *times* the number of contexts in which they appear.

The reviewer’s work can be made substantially easier by breaking up the messages into parts. The candidate’s name can be recorded and stored once, then used for all the messages that have to do with that candidate. The remaining part (in our example, “has been selected”) can be recorded once and used for all the selection messages across all candidates. The consistent reuse of audio clips can be checked mechanically, leaving the human reviewer with fewer audio clips to review (roughly the number of candidates *plus* the number of contexts).

Differences between Pvote and Ptouch

In order to support synchronized audio and video, Pvote's ballot definition format is substantially more complex than that of Ptouch. Figure 7.1 presents a side-by-side comparison of the ballot definition formats for Ptouch and Pvote. Only the main part of the ballot definition, the *ballot model*, is shown.

The rest of this section describes some of the main differences. In the terminology used here, a **contest** is a race or a referendum put to the voters and an **option** is one of the choices available in a contest. The options in a race are candidates, whereas the options in a referendum are typically “yes” and “no.” During voting, the **selection state** is the voter's current set of selections in all the contests. A contest is said to be **empty** if none of its options are selected, and **full** if the maximum allowed number of selections is selected. The **capacity** of a contest is its maximum allowed number of selections. To **undervote** in a contest is to leave the contest less than full; to **overvote** in a contest is to exceed its capacity.



Pages contain states. Pvote adds *states* within pages to represent a second level of focus, which is necessary to support navigation for blind users. Because audio navigation units are finer-grained, audio information is primarily specified at the state level, whereas visual information is primarily specified at the page level. All the states belonging to a page share the same overall appearance and layout, though a part of the screen can vary in appearance. Behaviours in response to user input can be specified at either level; at the state level they apply to a single state; at the page level they apply to all the states in the page.

For example, in a typical ballot layout, a single page presents a list of candidates, and each state within that page highlights one of the candidates. The user presses a button to step through the candidates one at a time. In the state when a particular candidate becomes the focus, the audio for the candidate's name is played and the candidate's name is

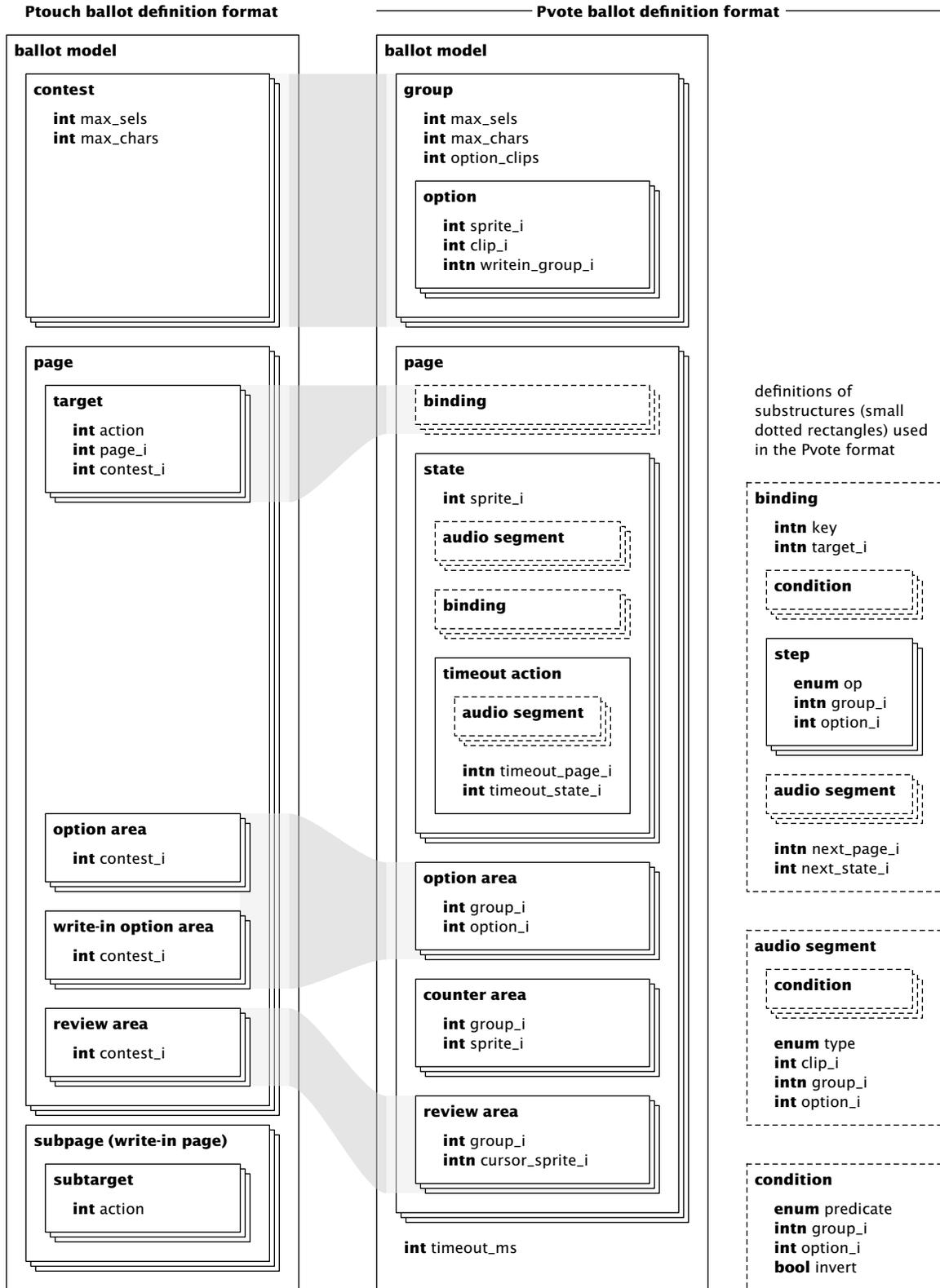
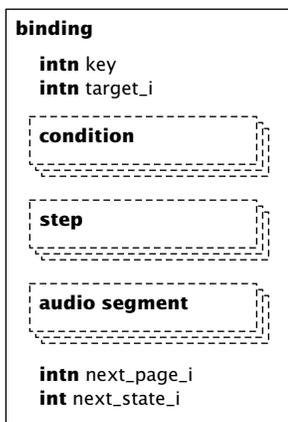


Figure 7.1. Comparison of Ptouch and Pvote ballot formats (only the ballot model is shown).

highlighted in the list on the screen. Selecting the currently highlighted candidate is a state-level behaviour, since the selection operation is different in each state, whereas moving on to the next contest is a page-level behaviour.

To help keep the user oriented, each state has a timeout audio sequence and an optional timeout transition. The ballot definition as a whole has a timeout parameter in milliseconds. When there has been no audio playing and no user input for the timeout period, the timeout audio sequence is automatically played and the timeout transition takes place, if any.



User inputs can be mapped to arbitrary actions. In the Ptouch format, the behaviours triggered by screen touches were specialized according to the type of the touched screen region. For example, option areas were hardcoded in the VM to react to a touch by toggling whether the associated option was selected, and write-in option areas were hardcoded to react to a touch by jumping to an associated write-in page.

This direct binding between screen regions and actions is inadequate for a multimodal design in several ways. First, direct binding doesn't make sense for input from hardware buttons: there aren't enough buttons to dedicate a button to each option. Second, the multimodal design has to allow for a "Where am I?" button, which could play many different audio messages depending on the current system state.

Third, text entry in an audio-only interface is a nontrivial design problem. Ptouch could afford to hardcode text entry behaviour in the obvious way—a keyboard made of onscreen buttons, where touching each button types a letter. But there is no single obvious way to enter text in an audio-only interface. For example, if the voting machine has space for a physical keyboard, then each key should type a letter. If the machine provides a button pad with "next", "previous", and "select" buttons, then the buttons could be used to navigate forward and backward through the alphabet to enter letters. The text entry method is likely to vary widely depending on the hardware, so it should be left up to the ballot definition to specify.

For all these reasons, Pvote allows more flexible input handling by adding a layer of indirection: a list of bindings between input events and the actions they trigger.

```
step  
enum op  
intn group_i  
int option_i
```

Actions are generalized to sequences of steps. With the introduction of bindings, there had to be a new data structure to represent the action triggered by an input event. An action is represented as a list of *steps*, where each step performs a selection operation (select an option, deselect an option, deselect the last selected option in a contest, or clear a contest). Actions with multiple steps are useful for straight-party voting and for ballots containing multiple versions of the same contests (e.g., large type and normal type). The list of steps is embedded in the data structure for a binding.

```
audio segment  
condition  
enum type  
int clip_i  
intn group_i  
int option_i
```

Audio sequences are attached to states and actions. Pvote can play audio when switching into a new state or when an action is triggered by user input. Also, when an action is triggered by user input, any currently playing audio is interrupted.

In the ballot definition, an audio sequence contains a list of audio segments, where each segment can be constant or variable. There are four kinds of variable audio segments:

1. A segment that plays the name of a specific option.
2. A segment that plays the names of all the selected options in a contest.
3. A segment that plays an audio clip chosen according to the current number of selected options in a contest.
4. A segment that plays an audio clip chosen according to the maximum number of selections a contest allows.

For example, to tell the voter which candidates are selected for city council, an audio sequence might consist of two segments: first a constant segment that says “Your selections for city council are”, then a variable segment that lists the voter’s selections in the city council contest. However, a constant segment is often insufficient to produce a natural-sounding description. If there is only one selection, the sentence should

begin “Your selection for city council is”. The third type of variable segment can be used to select the grammatically correct sentence.

The first and fourth types don’t vary depending on the selection state—any ballot that uses them can be defined just as well in a ballot definition language without them. But their presence allows more of the ballot definition to be kept the same from election to election, reducing the work of verifying the ballot definition.

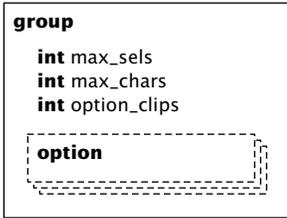
```
condition  
enum predicate  
intn group_i  
int option_i  
bool invert
```

Actions and audio segments can be conditional. Because Pvote’s behaviour in response to user input is no longer hardcoded, the ballot definition needs a way to specify different effects that will occur depending on the selection state. For example, consider what should happen when the user touches an option. If the option is already selected, then one possible effect would be to deselect the option. If the option is not selected, and its contest is not full, then the option should become selected. And if the option is not selected but its contest is full, then the selection should not change. Each of these three cases also needs its own corresponding audio message describing what happened.

To make this possible, each binding has an attached list of conditions concerning the selection state. Each condition can check whether a particular option is selected, a particular contest is full, or a particular contest is empty. The binding is triggerable only if all of its conditions are satisfied.

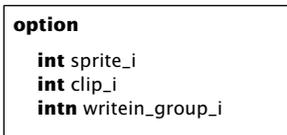
Conditions are also useful for constructing variable audio sequences. A list of conditions is attached to each segment; each segment is played or skipped depending on whether all of its conditions were satisfied. Reusing conditions in this way increases the flexibility of audio feedback while keeping the implementation simple.

Groups replace contests and write-ins. A *group* is a container of selectable options; it can represent a contest (with options such as candidates) or a write-in entry field (where the options



are the individual characters that can appear in the entry field). The group data structure is used for both purposes because of the functionality that is common between them:

- In both cases, the current selection for a group is a list of options (even though a contest selection has set-like semantics and a write-in selection has ordered sequence semantics).
- In both cases, user actions add and remove options to and from the selection (e.g., selecting candidates in a contest or typing letters into a write-in field).
- Visual display of the selections in a group consists of pasting the candidate images or the letter images into a sequence of equal-sized spaces on the screen.
- Audio playback of the selections in a group consists of playing each selection in order—reading off the list of selected candidates or speaking the letters in a write-in field one by one.



Options have their own data structure. In the Ptouch format, every option area was assumed to represent a distinct option. Thus, each option area only had to indicate which contest it belonged to. The Ptouch structure did not list the options in each contest; determining the number of options in a contest required scanning the pages of the ballot definition and counting the option areas associated with that contest.

In the Pvote format, information about each option—such as its associated image and audio clip—is kept in an *option* structure under the option’s group. The option areas refer to these option structures. Bindings that select options, audio segments that play option names, and conditions that examine options can either refer to options directly or refer to option areas, which themselves refer to options. This extra layer of indirection yields two kinds of flexibility:

- The same option can be displayed in more than one place on the ballot.
- Options can be rearranged by rearranging the references from option areas to options.

The rearrangement of options, also known as “candidate rotation,” helps to reduce the bias inherent in displaying a particular candidate first. Without the extra layer of indirection, candidate rotation would be difficult to automate reliably because there would be no distinction between a reference to an option area and a reference to an option. This distinction is important because indirect references to options via option areas should change when options are shuffled, whereas direct references to options should not change when options are shuffled. When candidates are rotated, their screen position and order of audio presentation should change, but the set of candidates belonging to a party for a straight-party vote should not change.

This design feature makes it easy to rotate candidates by a simple manipulation of the ballot file. Rearranging the references from option areas to options does not change the option number assigned to each candidate. Thus, candidate rotation has no effect on the way voter selections are recorded, which helps to avoid the possibility of confusion in interpreting recorded votes.

One could produce several rotated variants of a ballot before the election and publish them all; it is straightforward to verify that two ballot definition files represent the same ballot except for reordering of the candidates. Alternatively, the voting machine could even perform candidate rotation on the fly for each voter, though the Pvote implementation does not do this.

Ballot definition format

Figure 7.2 depicts the complete ballot definition format for Pvote. Just as in Ptouch, the ballot definition describes a state machine. Each state transition is triggered by a user action or by an idle timeout. Executing a transition can cause options to be selected or deselected. Audio feedback can be associated with states and with transitions between states. The ballot definition contains three main sections:

- *Ballot model*: structure of the ballot and interaction flow of the user interface.
- *Audio data*: sound clips to play over the headphones.
- *Video data*: images to display on the screen, the locations at which to display them, and locations of touch-sensitive screen regions.

These three sections are separated so that each one can be supplied to a distinct module of the VM with distinct responsibilities. In addition, they can be separately updated—for example, one can translate the audio interface into a different language by recording audio clips for a new audio data section while leaving the other sections unchanged.

In Pvote, which is written specifically for a text-based electronic ballot printer, the ballot definition also includes a fourth section, the *text data*, which contains textual descriptions of the contests and candidates for the printer to print.

Audio data. The audio data section specifies the sample rate at which all audio is to be played and provides an array of sound clips. Other parts of the ballot definition refer to these clips by supplying indices into this array. The audio clips are uncompressed and monophonic, and each sample is a 16-bit signed integer. The clips can contain recordings of actual speech or of prerendered synthesized speech.

Video data. The video data section specifies the resolution of the video screen and includes an array of *layouts* and an array

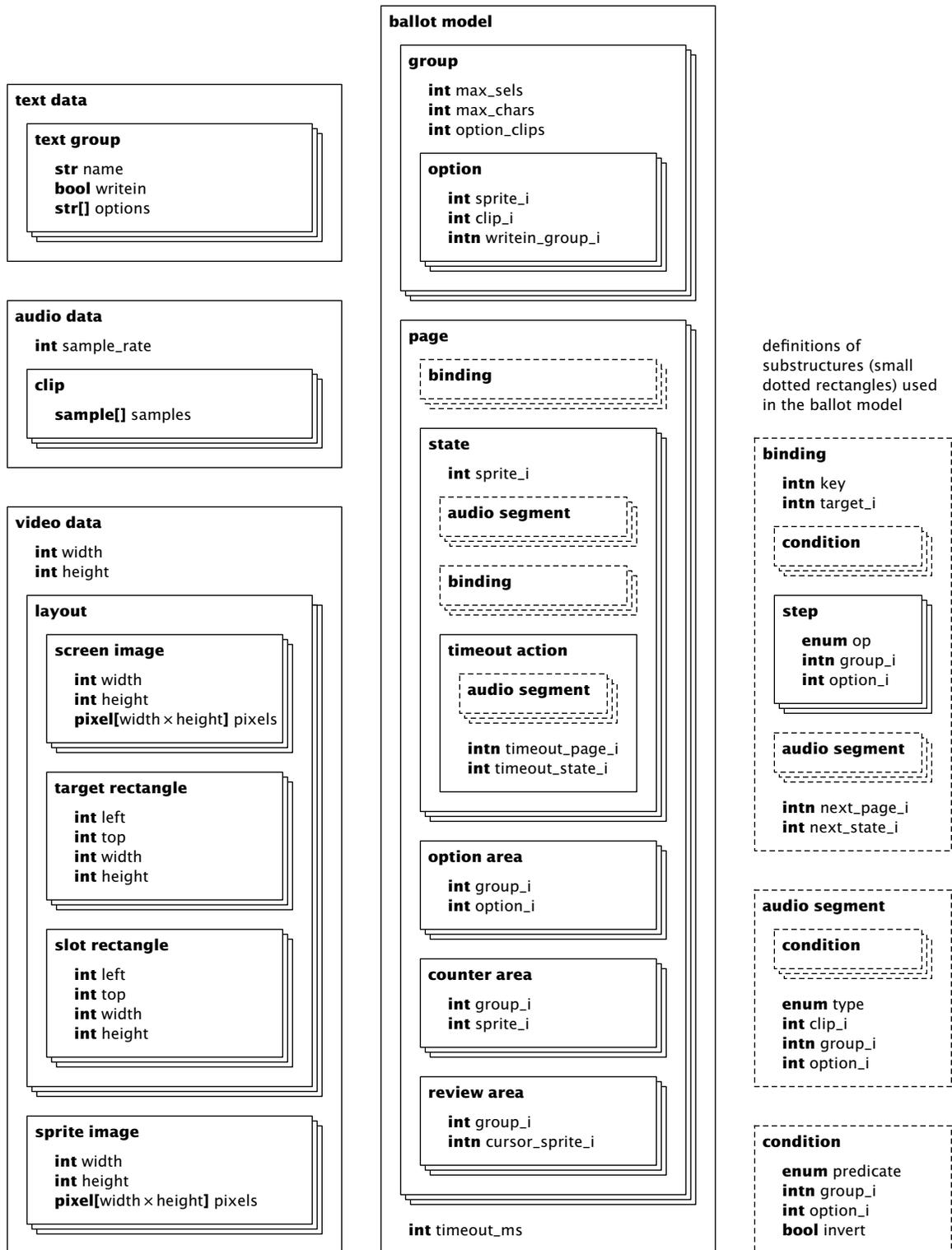
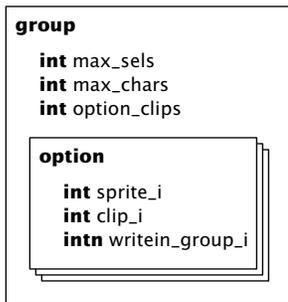


Figure 7.2. The Pvote ballot definition data structure. Stacked boxes represent arrays. This is the second line of the caption.

of *sprites*. A **sprite** is an image, smaller than the size of the entire screen, that will be pasted on the screen somewhere. A **layout** consists of a full-screen image, an array of *targets*, and an array of *slots*. A **target** is a rectangular region of the screen where a touch will have an effect; a **slot** is a rectangular region where a sprite can be pasted. Image data is stored uncompressed, with 3 bytes per pixel (red, green, and blue colour values).

Ballot model. The ballot model is the main specification of the state machine. It contains an array of *groups* and an array of *pages*. It also specifies an idle timeout in milliseconds.

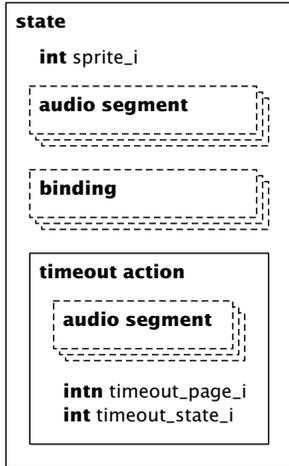


Groups and options. A **group** is a set of choices from which the voter makes selections. There are two kinds of groups: *contest groups* and *write-in groups*. A **contest group** represents a race in which the options are candidates or a referendum question with options such as “yes” and “no”. A **write-in group** represents the text entered in a write-in area within a contest, in which the options are the characters used to spell out the name of the write-in candidate. In the array of options within each group, images and sound clips are specified to represent each option by providing indices into the arrays of audio clips and sprites. Within a contest group, an option can also specify that it is a write-in option and identify the write-in group containing its write-in text.

Each group specifies its capacity (the maximum number of selections allowed in the group); for contest groups this prevents overvotes, and for write-in groups this limits the length of the entered text. All the write-in options within a contest must have the same maximum length for text entry.

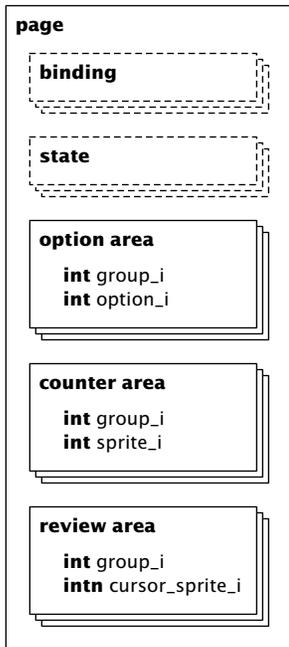
Pages and states. The **page** is the basic unit of visual presentation; within each page is an array of *states*. The pages correspond, one-to-one, to the layouts in the video data. At any given moment, there is a current page and a current state. The user interface always begins on page 0 in state 0; when the VM

executes a transition to the last page in the array of pages, the ballot is printed or cast with the voter's current selections. In addition to the array of states, each page contains arrays of *option areas*, *counter areas*, *review areas*, and *bindings*.



The **states** in a page are states in the state machine of the user interface. Each state specifies a sprite to be pasted over the main page image while the state is current. (For example, a page could show a list of several options, and the states within that page could display a focus highlight that moves from option to option. Each state would paste a focus highlight for its option over the page image.) Each state also has an array of *audio segments* to be played upon entering the state, and an array of its own bindings.

A state can also specify audio segments to be played upon a timeout and/or an automatic transition to another state upon a timeout. A timeout occurs when the audio has stopped playing and there has been no user activity for the timeout duration specified in the ballot model.



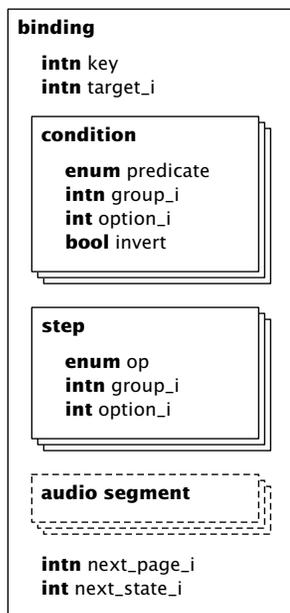
An **option area** is a screen region where an option will be displayed. Its fields identify the option that will appear there.

A **counter area** is a screen region that will indicate the number of options currently selected in a contest; this enables the interface to provide feedback on undervoting. A counter area is associated with a group and points to an array of sprites. The number of currently selected options in the group is used as an index to select a sprite from the array to display.

A **review area** is a screen region where currently selected options will be listed; it has a field to indicate the group whose selections will be shown. The review area must provide enough room for up to j options to be displayed, where j is the capacity of the group. A review area can also specify a “cursor sprite” to be displayed in the space for the next option when the group is not full. This allows a review area for a write-in group to serve as a text entry area, in which a cursor appears in the space where the next character will be added.

The screen locations for pasting all these sprites (overlays for states, options for option areas and review areas, and sprites

for counter areas) are not given in the ballot model; they are specified in the array of *slots* in the page's corresponding layout. Each state, option area, and counter area uses one slot. Each review area uses $j \times (1 + k)$ slots, where j is the capacity of the group and k is the capacity of write-ins for options in the group. (A write-in group cannot itself contain write-in options; thus, for a review area for a write-in group, k is zero.) Each block of $1 + k$ slots is used to display a selected option: the option's sprite goes in the first slot, and if the option is a write-in, the characters of the entered text go in the remaining k slots, which are typically positioned within the first slot. If there are i currently selected options in the group, option sprites appear in the first i of the j blocks. If there is a cursor sprite, it is pasted into the first slot of block $i + 1$ when the group is not full.



Bindings. The lists of bindings in pages and states specify behaviour in response to user input. Each binding consists of three parts: stimulus, conditions, and response.

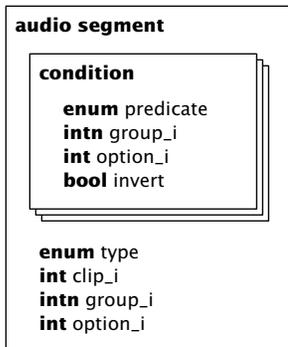
There are two kinds of stimuli: a keypress, which is represented as an integer key code, and a screen touch, which is translated into a target index by looking up the screen coordinates of the touch point in the layout's list of targets. A binding can specify either a key code or a target index or both.

Each binding can have a list of associated conditions; the binding applies only if all the conditions are satisfied. A condition can test whether a particular group is empty or full or whether a particular option is selected.

The response consists of three parts, all optional: selection operations, audio feedback, and navigation. The selection operations are specified as a series of *steps*, where a step selects or deselects an option, appends a character to a write-in, deletes the last character, or clears a group. The audio feedback is given as an array of audio segments to play. Navigation is specified as the index of a new page and state.

Bindings for the current state take precedence over bindings for the current page. When the user provides a stimulus, at most one binding is invoked: the bindings for the state and

then the page are scanned in order, and the response is carried out for the first binding that matches the stimulus and has all its conditions satisfied.



Audio segments. Audio feedback is specified as a list of segments. A segment can play a fixed clip, the clip associated with a specified option, all the clips associated with the options that are selected in a specified group, or a clip chosen based on the number of options that are selected in a specified group. When a clip associated with an option is played, if the option is a write-in option, the clip for each character in the contents of the write-in field is also played. More than one clip can be associated with an option (for example, each candidate could have a short description and a long description).

At any given moment, at most one clip can be playing at a time; there is a play queue for clips waiting to be played next. Whenever a clip finishes playing, the next clip from the queue immediately begins to play, until the queue is empty. Invoking a binding always interrupts any currently playing clip and clears the play queue. The audio segments for the binding, if any, are queued first; if a state transition occurs, the audio segments for the newly entered state are queued next.

Each segment has a list of conditions (the same as in a binding) that must all be satisfied in order for the segment to be queued; otherwise, the segment is skipped. The conditions are evaluated when the segment list is being queued (i.e., immediately after carrying out the selection steps of a binding, immediately after entering a new state, or when a timeout occurs).

Software design

The virtual machine is composed of five software modules: the *navigator*, the *audio driver*, the *video driver*, the *event loop*, and the *vote recorder* (Figure 7.3). Each component has limited responsibilities, and there are limited data flows between components. Two additional components not visible in Figure 7.3 are the *ballot loader*, which deserializes the ballot definition into memory, and the *ballot verifier*, which checks the ballot definition. The loader and verifier complete their work before the voting session begins (i.e., before any interaction with the voter). The verifier is responsible for ensuring that the ballot definition is sufficiently well-formed that the VM will not crash or become unresponsive during the voting session.

The **event loop** maintains no state and handles all incoming events, which are of four types:

- **Keypresses:** Upon receiving a keypress event, the event loop sends a **press** message to the navigator.

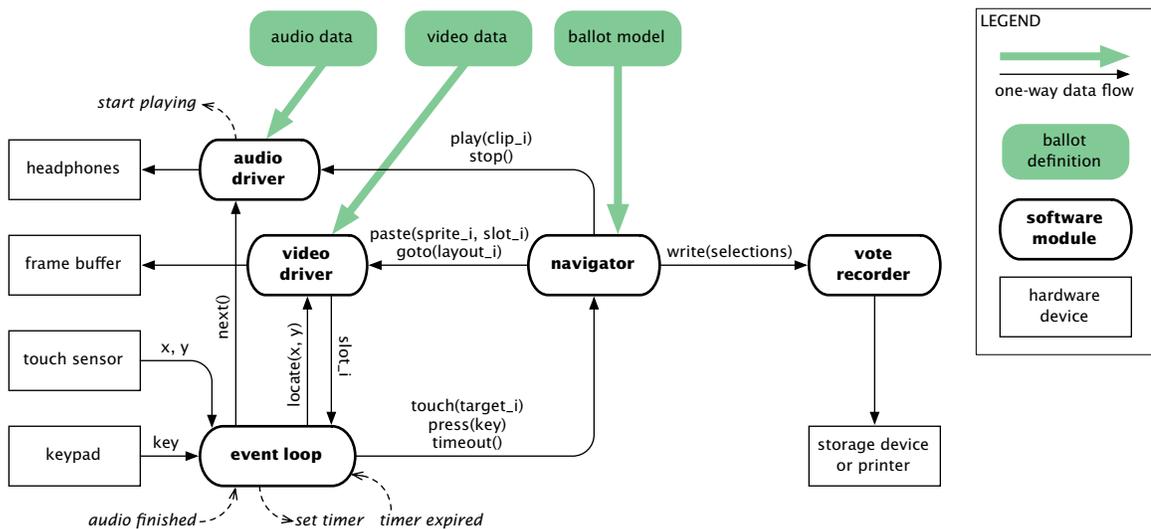


Figure 7.3. Block diagram of the Pvote virtual machine. The five software modules in bold generate and run the user interface. The arguments `clip_i`, `layout_i`, `sprite_i`, `target_i`, `key`, `x`, and `y` are integers; `selections` is an array of arrays of integers.

- Screen touches: Upon receiving a touch event, the event loop sends a **locate** message to the video driver to translate the touch coordinates into a target index, then passes this target index to the navigator in a **touch** message.
- Audio notifications: Upon receiving notification that a sound clip has finished playing, the event loop sends a **next** message to the audio driver.
- Timer notifications: Upon receiving notification that the timer has expired, if no sound clip is currently playing, the event loop sends a **timeout** message to the navigator to indicate that the ballot's specified timeout has passed with no activity.

Whenever it receives any event, the event loop reschedules a timer notification event according to the timeout duration in the ballot definition.

The **navigator** keeps track of the current page and state and the current selections in each group, and has no other state. The navigator responds to three messages:

- **touch(target_i)**: Find the first operative binding for the

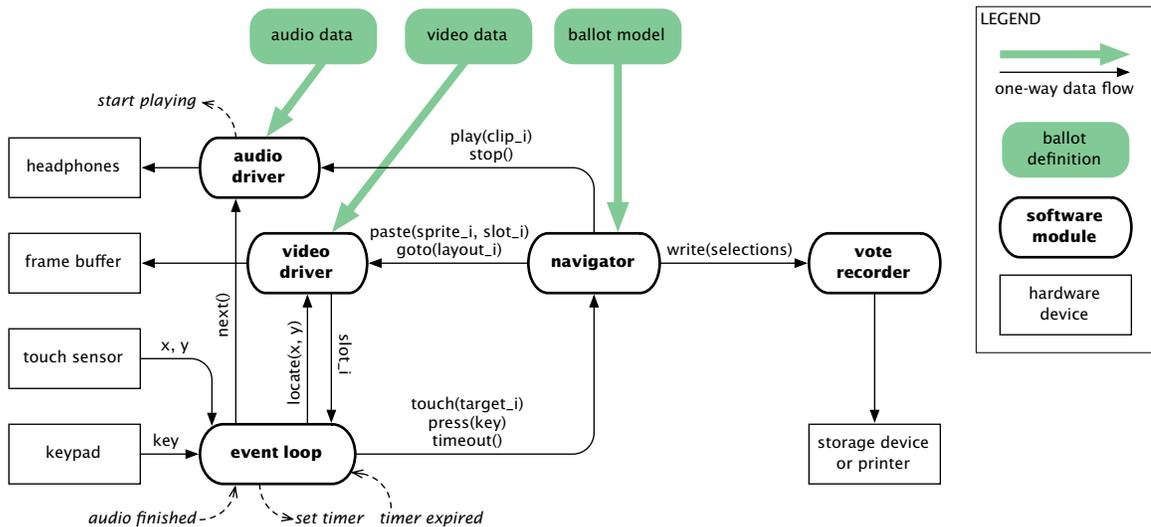


Figure 7.3. Block diagram of the Pvote virtual machine. The five software modules in bold generate and run the user interface. The arguments `clip_i`, `layout_i`, `sprite_i`, `target_i`, `key`, `x`, and `y` are integers; `selections` is an array of arrays of integers.

current state or page that matches the given target, and invoke it.

- **press(key)**: Find the first operative binding for the current state or page that matches the given keypress, and invoke it.
- **timeout()**: Add the current state's timeout audio segments to the play queue, and follow the current state's timeout transition, if one is specified.

The navigator sends five messages to other modules:

- **goto(layout_i)** is sent to the video driver upon transition to a page. The layout index is the same as the page index (the array of layouts in the video data parallels the array of pages in the ballot model).
- **paste(sprite_i, slot_i)** is sent to the video driver to paste sprites into slots as necessary for states, option areas, counter areas, and review areas. *sprite_i* is the index of a sprite in the array of sprites in the video data; *slot_i* is the index of a slot in the current layout.
- **play(clip_i)** is sent to the audio driver to queue a clip to be played on the headphones. *clip_i* is the index of an audio clip in the array of clips in the audio data.

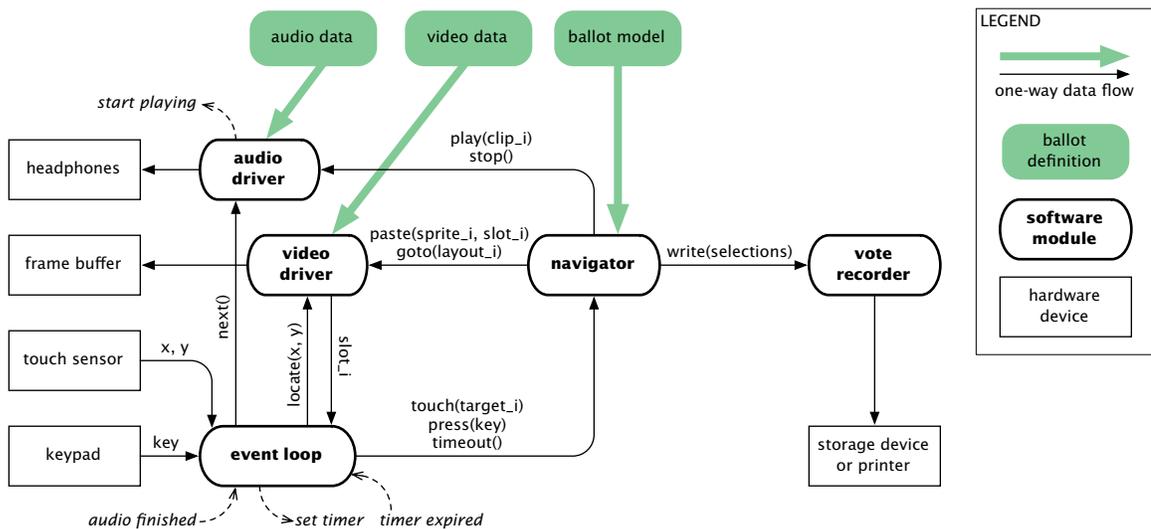


Figure 7.3. Block diagram of the Pvote virtual machine. The five software modules in bold generate and run the user interface. The arguments *clip_i*, *layout_i*, *sprite_i*, *target_i*, *key*, *x*, and *y* are integers; *selections* is an array of arrays of integers.

- **stop()** is sent to the audio driver to stop the currently playing clip.
- **write(selections)** is sent to the vote recorder to record the user's selections. `selections` is an array of arrays of integers: one array for each group, listing the indices of the selected options in that group.

The **audio driver** maintains a queue of audio clips to be played, and has no other state. It responds to three messages:

- **play(clip_i)**: If nothing is currently playing, immediately begin playing the specified clip; otherwise queue the specified clip to be played.
- **next()**: If there are any clips waiting in the queue, start playing the next one.
- **stop()**: Stop whatever is currently playing and clear the queue.

The audio driver sends no messages to other modules, but whenever it starts playing a clip, it schedules a notification event for the event loop to receive when the clip finishes playing. The audio driver also exposes a flag that the event loop reads to check whether audio is currently being played.

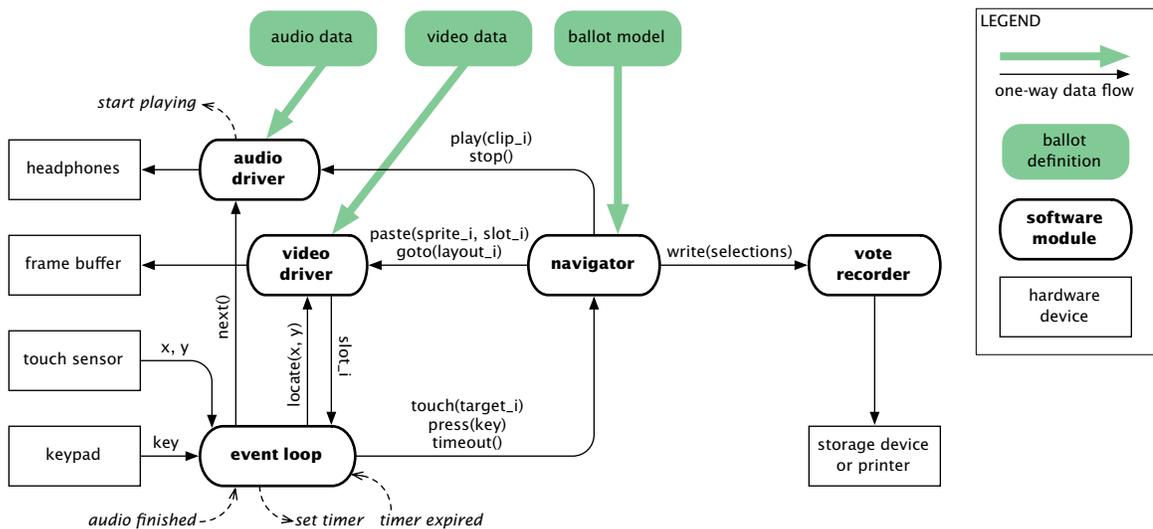


Figure 7.3. Block diagram of the Pvote virtual machine. The five software modules in bold generate and run the user interface. The arguments `clip_i`, `layout_i`, `sprite_i`, `target_i`, `key`, `x`, and `y` are integers; `selections` is an array of arrays of integers.

The **video driver** maintains just one piece of state, the index of the current layout. It responds to three messages:

- **goto**(layout_i): Copy the full-screen image for the given layout into the video display's frame buffer and remember this as the current layout.
- **paste**(sprite_i, slot_i): Copy the given sprite into the frame buffer at the position specified by the given slot in the current layout.
- **locate**(x, y): Find and return the index of the first target that contains the given point in the current layout's list of targets, or an error code if the point does not fall within any target.

The video driver sends no messages to other modules.

The **vote recorder** maintains no state and responds to only one message:

- **write**(selections): Record the voter's selections.

The vote recorder records votes as appropriate for the type of voting machine (e.g., printing a ballot, marking a ballot, or directly recording votes in electronic storage).

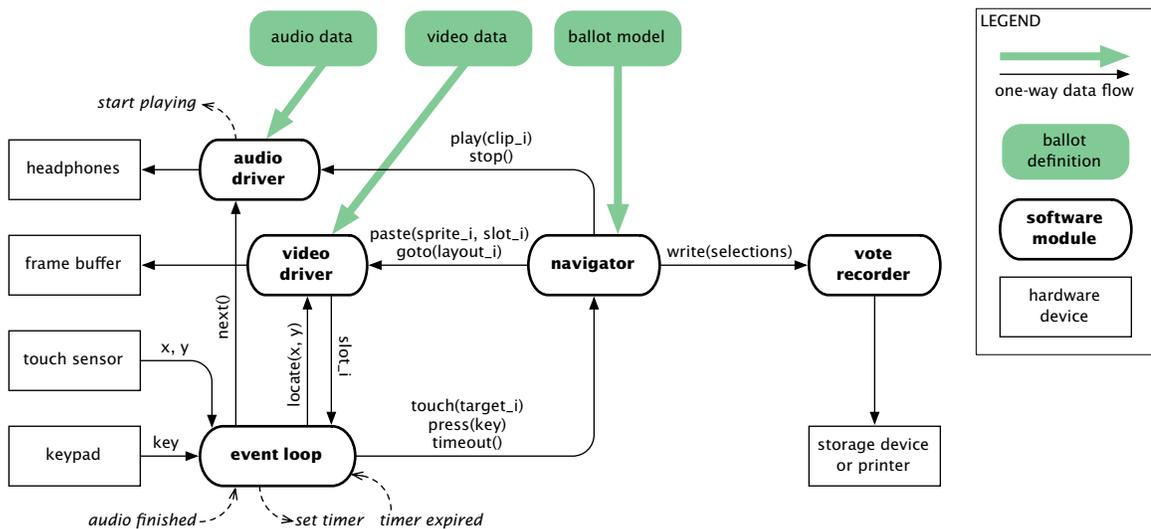


Figure 7.3. Block diagram of the Pvote virtual machine. The five software modules in bold generate and run the user interface. The arguments clip_i, layout_i, sprite_i, target_i, key, x, and y are integers; selections is an array of arrays of integers.

Implementation

Pvote is a Python [63] implementation of the design described here. Pvote can run on Linux, MacOS, and Windows. Graphics and sound are handled by Pygame [62], an open-source multimedia library for Python. Touchscreen input is simulated using the mouse, and hardware button input is simulated using the keyboard.

Pvote is written to be deployed as an electronic ballot printer. In Pvote, the vote recorder prints out a textual description of the voter's selections. Each time Pvote runs, it prints at most one ballot (to standard output) and then enters a terminal state. The source code for Pvote is included in Appendix B. The code is also available online at <http://pvote.org/>, together with a sample ballot definition file in the Pvote format. The sample ballot definition is described in detail in Appendix C.

Evaluation

Size. The entire Pvote implementation is 460 lines long, not counting comments and blank lines. The breakdown of module sizes is as follows:

ballot loader	137 lines	
ballot verifier	96 lines	
<hr/>		
subtotal (pre-voting)		233 lines
event loop	25 lines	
navigator	120 lines	
audio driver	35 lines	
video driver	22 lines	
<hr/>		
subtotal (voting)		202 lines
vote recorder		25 lines
<hr/>		
total		460 lines

Dependencies. Pvote is written in a small subset of Python 2.3, called Pthin, which is specified in the Pvote Assurance Document [92]. Pvote uses only one built-in collection type, the Python list, and only the following built-in functions:

- `open` and `read` to read the ballot definition file.
- `chr` and `ord` to convert integers to/from characters.
- `list` to convert strings to lists of characters.
- `enumerate` and `range` to iterate over lists.
- `len`, `append`, `remove`, and `pop` to manipulate lists.

The ballot loader imports the built-in SHA module and uses it to verify a SHA-1 hash of the ballot definition. The audio and video driver use various Pygame functions: `init` and `stop` in the audio mixer module, `play` on the **Sound** object, `init` and `set_mode` in the video display module, `fromstring` in the image module for loading images, and `blit` on the **Surface** object to paste images onto the screen. Aside from these, Pvote imports no other library modules.

File size. Pvote was tested with a sample ballot definition file generated by a ballot compiler, also written in Python. The ballot compiler takes a textual description of the contests and options and produces the necessary images using the open-source ReportLab toolkit [65] for drawing, text rendering, and page layout. To construct the audio clips for the ballot definition, the compiler uses the same textual description to select fragments from a library of clips of recorded speech and concatenates the fragments together as needed. The audio clips in this sample ballot are recorded from live speech, which is usually preferred over synthesized speech.¹

The inclusion of screen images and audio recordings in the ballot definition yields a large file. See Appendix C for details on the sample ballot. It contains five contests: two are single-selection races with six candidates each, one is a multiple-selection race with five candidates, and two are propositions. An audio description of about 100 words for each proposition is included in the ballot. The result is a 69-megabyte ballot definition file, containing 17 pages at a resolution of 1024×768 pixels and 8 minutes of audio sampled at 22050 Hz. As a rough estimate, a ballot with 20 or 30 contests might occupy a few hundred megabytes.

File sizes this large might seem unwieldy in practice. However, files can be compressed for transmission (bzip2 compresses this 69-megabyte ballot to 12.5 megabytes, which is better than a factor of 5), and ballot definitions can be loaded onto voting machines using inexpensive SD flash memory cards (one-gigabyte SD cards can be purchased for about US\$10).

Functionality. Pvote achieves the functionality goals that were listed at the end of Chapter 6. Pvote can support a wide range of features in the voting user interface, including multimodal input and output and virtually complete flexibility in the style of audio and visual presentation. Because Pvote uses

¹The National Council on Disability wrote, “Voting systems that provide digitized human speech are preferable to systems with synthesized speech because digitized speech is ‘more readily comprehensible’ and more likely to contain the correct pronunciation of candidate names” [51].

prerecorded audio and prerendered images, the ballot can be presented in any language.

With its generalized actions and conditions, Pvote offers much more flexibility in the handling of user input than Ptouch, its touchscreen-only predecessor. Unlike Ptouch, Pvote can handle straight-party voting, dependencies among contests (e.g., in a recall election, voting for a replacement candidate conditional on voting “yes” for recalling the incumbent), and conditional navigation (e.g., displaying an undervote warning page when the voter has not made any selections in a contest). The ballot designer also has more freedom to define the interaction for selection and text entry.

To get a rough sense of Pvote’s coverage of ballot design features, I examined NIST’s collection of sample ballots [56], consisting of 373 ballots from 40 U. S. states for elections from 1998 to 2006. The longest was a 2004 ballot from Chicago that had 15 elected offices, 74 judicial confirmations, and one referendum. The following table summarizes the features used on these ballots. All these features, and hence all the ballots in the collection, are supported by Pvote’s ballot definition format.

Ballot feature	Ballots
Vote for 1 of n	373
Vote for up to k of n ($k > 1$)	195
Vote for an image (e.g., a state flag)	2
Vote yes/no (referendum, confirmation)	251
Ranked choice (up to 3 choices)	7
Write-in candidate	318
Straight-party vote	60
Cross-endorsed candidates	8
Multi-party primary	5
Party logos	21

The collection also includes ballots in Chinese, Ilokano, Japanese, Korean, Spanish, and Vietnamese. Pvote can present ballots in any language, though for write-in candidates voters must spell out the name using an alphabetic language.

8 Security review

How was Pvote's security evaluated?	137
What were Pvote's security claims?	139
How was Pthin defined?	143
What flaws did the reviewers find?	145
What improvements did the reviewers suggest?	146
Did the reviewers find the inserted bugs?	148
What ideas did reviewers have on programming languages?	149
What ideas did reviewers have on conducting reviews?	151
What lessons were learned from the review?	153

How was Pvote's security evaluated?

My overall purpose in creating Pvote was to design and write voting software whose security could be easily verified. To test whether it had achieved this purpose, I invited several security researchers to all-day meetings at the University of California, Berkeley to review the Pvote design and source code. Reviewers met from March 29 to March 31, 2007 and also on May 20, 2007.

David Wagner and I were on hand for all three days in March to explain Pvote's design, answer the reviewers' questions, and provide any assistance they requested in their investigation. On May 20, I attended but David Wagner did not.

The reviewers examined and discussed Pvote for a total of about 90 reviewer-hours over the four days of reviewing.

Participants. On March 29 and 30, these reviewers were present:

- Matt Bishop, UC Davis
- Mark Miller, HP Labs
- Dan Sandler, Rice University
- Dan Wallach, Rice University

On March 31, these reviewers were present:

- Tadayoshi Kohno, University of Washington
- Mark Miller, HP Labs
- Dan Sandler, Rice University

On May 20, these reviewers were present:

- Ian Goldberg, University of Waterloo
- Tadayoshi Kohno, University of Washington

The assurance document. Before the review, I prepared a 77-page document to provide the reviewers with detailed information about Pvote. This document [92] presents the ballot definition format, the software design, and the source code of Pvote itself. The source code is displayed with annotations justifying the validity of each line, shown on the facing page opposite each page of code.

Not all the reviewers had previous experience with the Python programming language. To ensure that everyone had a common understanding of the code, I had to provide a definition of the language in which it was written. I chose to define a small subset of Python called Pthin, containing just the syntactic constructs and functions used by Pvote. With the language semantics clearly specified, we could exclude flaws in the language implementation from the security review, and focus on Pvote itself.

The assurance document defined the scope of the review by stating assumptions about how Pvote would be used and listing the security properties that Pvote was supposed to uphold under those conditions. These properties were drawn from the assurance tree given in Chapter 2 and the security goals given in Chapter 6. For each claimed security property, I gave an assurance argument.

The review process. I spent most of the first day presenting the software design of Pvote and walking the reviewers through the implementation. For the rest of the first day and the second day, the reviewers examined the software, mostly by hand, and asked us questions. We discussed various aspects of Pvote, voting security, and software reviewing in general.

By the end of the second day, David Wagner and I realized that, because the reviewers had not found any bugs and we did not know of any bugs in the code, we could not conclude anything about how effective they were at finding bugs or whether any bugs were actually present. Therefore, to motivate the reviewers and observe their effectiveness at finding bugs, we decided to intentionally insert some bugs into the code. On the third and fourth days, we announced that the code contained at least one bug, and asked the reviewers to find it. On the fourth day we also asked the reviewers to try inserting their own bugs, hoping this would motivate them to understand the code in more depth.

What were Pvote's security claims?

Pvote was evaluated against a set of *responsibilities*, under a set of *assumptions* about how it is deployed for an election. Both of these are listed below.

Since several possible vote-recording mechanisms can be used with Pvote, I had to coin a generic term to refer to the recording step. Thus, the term **committed** means that voter selections are finalized as far as the machine is concerned—this occurs on a DRE when votes are recorded, but on an EBM or EBP when votes are printed. The following lists also use the term **voting session**, which lasts from when a voting machine starts interacting with a particular voter (e.g., when the first screen of the voting user interface comes up) until the ballot is committed or the voter abandons the machine. This does not include per-voter initialization steps by pollworkers.

Assumptions. The reviewers were asked to assume that:

- A1. The voting machine software (ostensibly Pvote) is handed over for review before the election.
- A2. The software that runs on the voting machines on election day is exactly what was reviewed.
- A3. Pvote is started once per voting session.
- A4. Only authorized voters are allowed to carry out voting sessions.
- A5. Ballot definition files are published for review and testing before the election.
- A6. The correct ballot definition is selected and used for each voting session.
- A7. The ballot definitions used on election day are intact, exactly as they were reviewed.
- A8. The programming language implementation functions correctly.
- A9. The operating system and software libraries function correctly.
- A10. The voting machine hardware functions correctly.

Responsibilities. Under the above conditions, Pvote must:

- R1. Never abort during a voting session. (For any given ballot definition, Pvote should either (a) always reject it as invalid and never start voting sessions, or (b) always accept it as valid and never abort during any session with that ballot definition.)
- R2. Remain responsive during a voting session.
- R3. Become inert after a ballot is committed.
- R4. Display a completion screen when and only when a ballot is committed, and continue to display this screen until the next session begins.
- R5. Exhibit behaviour in each session independent of any previous sessions.
- R6. Exhibit behaviour independent of which parts of buttons are touched (all touch points within a target region should be equivalent).
- R7. Exhibit behaviour that is determined entirely by the ballot definition and the stream of user input events and their timing.
- R8. Commit valid selections (no overvotes and no invalid candidates or contests).
- R9. Commit the ballot when and only when so requested by the voter.
- R10. Correctly and unambiguously commit the selections the voter made.
- R11. Present instructions, contests, and options as specified by the ballot definition.
- R12. Navigate among instructions, contests, and options as specified by the ballot definition.
- R13. Select and deselect options according to user actions as specified by the ballot definition.
- R14. Correctly indicate which options are selected, when directed to do so by the ballot definition.
- R15. Correctly indicate whether options are selected, when directed to do so by the ballot definition.
- R16. Correctly indicate how many options are selected, when directed to do so by the ballot definition.

Examples of threats. The above set of assumptions placed certain threats out of scope for the review, such as:

- *Insiders among pollworkers.* We assumed that pollworkers would not give voters multiple sessions (A3), would not let unauthorized people vote (A4), and would select the correct ballot style for each voter (A6).
- *Tampering with the software distribution.* We assumed that the voting machine software would not be altered between review and use (A1, A2).
- *Tampering with the ballot definition.* We assumed that the ballot definition would not be altered between review and use (A5, A7).
- *Tampering with cast vote records.* We assumed that other mechanisms would protect the integrity of paper or electronic vote records produced by Pvote.
- *Faulty or subverted non-voting-specific software.* We assumed that the software components that are not specific to voting function correctly (A8, A9). The assurance document describes the proper behaviour of the library functions and operating system.
- *Faulty or subverted hardware.* The review focused only on software (A10).
- *Poor ballot design.* It was specifically not claimed that using Pvote would eliminate accessibility or usability problems, even though testing with the published ballot definitions might help reveal some of these problems in time to address them.

The review focused on threats of the following four kinds:

- *Voters.* Voters can interact with Pvote using the touchscreen and keypad. Is there any sequence of interactions that can cause Pvote to violate voting rules (R3, R4, R8) or violate voter privacy (R5)?
- *Bugs.* Can any valid ballot definition, in combination with any sequence of user interactions, ever cause Pvote to behave incorrectly (R1, R2, R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, R16)?

- *Insiders among voting software suppliers.* An insider might modify Pvote to contain backdoors or hidden weaknesses before being handed over for review and installation. Could an attacker make effective changes that would go unnoticed by reviewers and testers?
- *Insiders among election officials.* An insider might design or alter a ballot definition to contain the wrong information or bias the vote. Could an attacker subvert ballot definitions in a way that would go unnoticed by reviewers and testers?

Insider threats were an area of particular attention because Pvote was designed specifically to address the problem that software is complex and hard to trust. One of the things I hoped to learn from the review was the effect of Pvote's novel design approach on the difficulty of performing or detecting an insider attack.

How was Pthin defined?

Pthin is a subset of the Python language; that is, all Pthin programs are valid Python programs. The following is just an overview of the Python features that are included in Pthin, for readers familiar with Python. For a complete Pthin specification, see the assurance document [92].

Features. In Pthin, values have types, but variables do not; any variable can be assigned a value of any type. There is a unique special value called `None` whose only supported operation is comparison to `None`. Aside from this, there are six types of values in Pthin:

- *Integers* are signed and unlimited in size.
- *Strings* contain 8-bit bytes.
- *Lists* have variable length and can contain values of any type as elements.
- *Functions* may take arguments of any type and always return a value (which is `None` if no value is explicitly returned).
- *Classes* contain method definitions; invoking a class (like a function) instantiates an object.
- *Objects* are instances of classes. Each object has its own public namespace of fields, accessed with a dot.

Pthin includes the following operators from Python:

- `=` for assignment to variables and object fields
- `.` for accessing object fields (as in `x.y = 5`)
- `+`, `-`, `*`, `/`, `%` for doing arithmetic on integers
- `+` for concatenating strings or lists
- `[]` for indexing strings and lists (as in `x[3]`)
- `[:]` for slicing strings and lists (as in `x[i:j]`)
- `==`, `!=`, `<`, `<=`, `>`, `>=` for comparing integers
- `==`, `!=` for comparing strings and comparing to `None`
- `and`, `or`, `not` for Boolean operations (these accept operands of any type and yield the integer values 0 or 1)
- `in` for testing if an element is in a list (as in `a in b`)

Pthin includes the following kinds of Python statements:

- `print` prints out a string
- `assert` causes a fatal error if a condition is not met
- `if` executes a block conditionally
- `for` iterates over the elements of a list
- `while` iterates on a condition
- `import` imports code from other modules
- `class` declares a class (but there is no inheritance in Pthin)
- `def` defines a function or a method
- `return` returns a value from a function

Pthin includes the following built-in Python functions:

- `range(i)` makes a list of the integers from 0 to $i - 1$
- `chr(i)` converts an integer to a one-byte string
- `ord(s)` converts the first byte of a string to an integer
- `len(x)` gets the length of a string or list
- `list(s)` breaks a string into a list of one-byte strings
- `enumerate(l)` turns a list `l` into a list of `[i, x]` pairs for each element `x` and its index `i`
- `open(s)` opens a file for reading

Pthin lists support the `append()`, `remove()`, and `pop()` methods from Python. Pthin includes *list comprehension* expressions, of the form `[x*x for x in range(5)]`, which evaluate an expression once for each element of a list to yield a new list containing all the results.

Properties. Pthin is a completely deterministic language, which is of critical significance for reviewing and testing. There is no access to clocks or sources of randomness. The only ways that a Pthin program can be influenced by the outside world are by reading from files and by receiving Pygame events.

The definition of Pthin eliminates some of the more complex features of Python, such as inheritance and exception handling. Exceptional conditions in Pthin cause fatal errors, since they cannot be caught.

What flaws did the reviewers find?

The reviewers did not find any bugs in the original Pvote source code. However, they did find some errors and omissions in the assurance document. I will describe the most significant ones here; all of the reviewers' findings are explained in detail in Appendix E.

Correctness claim for R1 (non-termination). Pvote is supposed to “never abort during a voting session” (R1), and the assurance document presents a supporting argument for this claim. The presented argument is incomplete because it neglects to rule out one way that Pvote could run out of memory. Nonetheless, it is still possible to show that memory usage has an upper limit; Appendix E provides the missing part of the argument.

Correctness claim for R9 (ballot casting). Pvote is supposed to “commit the ballot when and only when so requested by the voter” (R9). However, a ballot definition can direct Pvote to automatically cast the ballot (by jumping to the last page) after some amount of time has passed with no user activity, in violation of this requirement. One of the assumptions is that the ballot definition file must be checked before the election (A5). To ensure that R9 is met, the pre-election check has to ensure that no automatic transition goes to the last page.

Missing requirement for voter privacy. The assurance document doesn't state an explicit requirement for preserving the voter's privacy once his or her ballot has been committed. Pvote is restarted afresh for each new voter (A3), but what about the interval from when the voter walks away until the machine is reset? A ballot definition that displays the voter's selections on the last page (i.e., after committing the ballot) might violate the voter's privacy. So the pre-election check must also prohibit such ballot definitions; the assurance document neglected to make this clear.

What improvements did the reviewers suggest?

The following are the main recommendations on which all the reviewers could agree; Appendix E lists all their suggestions in more detail, including those that were less conclusive.

Assurance document. The reviewers recommended including a detailed breakdown of all the properties to be verified about the ballot definition, divided into three categories:

- properties checked by Pvote's verifier,
- properties checked by other automated tools, and
- properties checked by humans.

This would address two of the three flaws mentioned in the last section (the problem with the correctness claim for R9 and the voter privacy concern about the last page).

The reviewers also recommended:

- adding a section that enumerates all causal connectivity between Pvote and the outside world;
- stating explicit preconditions about the state of the audio driver when the navigator's `timeout()` method is called;
- mentioning that cursor sprites need to be checked to ensure they can't be confused with any option sprites or character sprites; and
- cautioning that, if an exception occurs during a voting session, Python will emit a stack trace that might reveal something about the voter's choices.

Pthin. The reviewers recommended these changes to Pthin, to simplify the language and facilitate reviewing:

- prohibiting all unprintable characters except newline;
- prohibiting all identifiers containing double-underscores, except `__init__`;
- prohibiting nested class or function definitions; and
- prohibiting chained assignments of the form `x = y = z`.

Ballot definition format. The reviewers recommended:

- offering ballot definition analysis tools to help reviewers check ballot definitions; (e.g., to ensure that all the pages can be reached from the starting page, to ensure that option areas don't overlap each other, and so on);
- defining an alternate textual representation of the ballot definition that is easier for humans to examine and edit, and providing tools to translate between the text form and the binary form;
- developing a translator that turns a ballot definition into a set of HTML pages or a Flash animation so that voters can preview the voting experience in a Web browser.
- renaming the **int** type to **nat** to make it clearer that no negative numbers are allowed, only natural numbers;
- placing digital signatures on ballot definitions and having Pvote check the signatures; and
- including the 8-byte file header in the input for computing the hash that appears at the end of the file.

Implementation. The reviewers recommended several changes to the Pvote code to improve its clarity and reviewability. Their suggestions and comments are described in the presentation of the code in Appendix B, as well as in Appendix E.

Did the reviewers find the inserted bugs?

David Wagner and I decided to insert three bugs into Pvote to see if the reviewers would find them. We inserted what we thought would be an “easy” bug, a “medium” bug, and a “hard bug” to find, and chose each bug individually in such a way that an insider could conceivably exploit the bug to influence the results of an election. These bugs are detailed in Appendix E.

We decided to insert all of these bugs in a 100-line region of a single file, lines 11 to 109 of `Navigator.py`, and told the reviewers to look in this region. We did this both because the navigator was the most interesting in terms of the program logic and because we knew the reviewers would have limited time. The new version of the code that we gave the reviewers contained all three bugs, but we did not tell the reviewers how many bugs there were.

Yoshi Kohno, Mark Miller, and Dan Sandler participated as reviewers on the third day of the review. Dan was very familiar with Python and found the “easy” and “medium” bugs quickly, within about 70 minutes. Yoshi Kohno and Mark Miller found the “easy” bug after about four hours of reviewing. None of the reviewers found the “hard” bug.

Ian Goldberg and Yoshi Kohno participated as reviewers on the fourth day of the review. Ian Goldberg also found the “easy” bug within about two hours; none of the other bugs were found on the fourth day.

The reviewers spent a total of about 20 reviewer-hours focused on the task of finding the bugs in this 100-line section of `Navigator.py`.

What ideas did reviewers have on programming languages?

The effect of programming language design on adversarial code review was a prominent topic of discussion. These are some of the main issues we discussed.

Mistyped or confusing identifiers. There are a few common ways that variable names and other identifiers can lead to problems in a software review:

- In Python, misspelled identifiers can lead to errors while the program is running.
- Identifiers that are too similar can confuse reviewers (intentionally or unintentionally).
- The same name can be used to refer to different things in different scopes.

We discussed several possible language restrictions that would help avoid these problems, such as requiring variable declarations, forbidding the shadowing of variables, forbidding the use of a field and a variable with the same name (e.g., `self.foo` and `foo`) in the same context, or forbidding variables with names that are too similar.

Language subsetting. Another way to reduce the burden on reviewers would be to let programmers choose restricted subsets of the language in which to write sections of the program. For example, suppose the programmer could declare that a particular function is written in a side-effect-free subset of the language, and a static verification tool could check that only allowed syntax is used. This restriction would make it easier for reviewers to audit the function and understand other functions that call it.

E [89] and Joe-E [45] are especially interesting examples of modern languages that support language subsetting, since they offer an *extensible auditing* feature that lets programmers define their own subsets of the language.

Static types. Types can be a powerful mechanism for statically checking program correctness. I chose to write Pvote in Python, a language without static type-checking, because of Python's agility and conciseness. On the other hand, static verification could have reduced some of the burden on reviewers at the cost of a longer and harder-to-read program.

Mutability. If the programming language supported a way of making variables immutable, this would be one fewer thing for reviewers to worry about (for example, the ballot definition could become immutable after it has been loaded and verified).

What ideas did reviewers have on conducting reviews?

Looking at source code. One reviewer remarked that he was much more effective at comprehending someone else's code when all the code was spread out on the wall in front of him, on paper. He found this surprising because he had spent the last 20 years editing code on computer screens.

This suggested to me that there might be significant value to keeping the code size below a threshold at which it is physically possible to lay out all of the code in front of a single person.

Trust in the adversary. The reviewers mentioned that it was difficult to maintain the requisite level of distrust in me as the author of the code, especially when we were interacting directly. On a few occasions, the reviewers found they were inclined to make unjustified assumptions about the good intent or competence of the author, and they later suggested that preventing social interaction between the reviewers and the author might make such reviews more effective.

Reviewer fatigue. The reviewers generally felt that the point where a reviewer becomes tired of inspecting a piece of code comes long before the code has been subjected to enough scrutiny. This suggests that it might be more effective for code to be reviewed by many reviewers each for a limited length of time, rather than a single reviewer for an extended length of time.

One-line change test. Mark Miller proposed a test for determining the size of the TCB (trusted computing base) for a particular security requirement—that is, the amount of code on which that requirement relies. His test consists of a series of trials with someone playing the role of the attacker. For each trial, one line of the program is chosen at random and the attacker is allowed to change just that line to do as much

damage as possible. The fraction of trials in which the attacker succeeds in violating the security requirement yields an estimate of the fraction of the program that constitutes the TCB for that requirement. Looking at the degree of vulnerability in these terms allowed us to talk about the potential value of a particular design change to Pvote or Pthin.

The read-write review. Dan Sandler proposed a new type of software review he called the “read-write review,” in which reviewers are asked to insert their own bugs. He conjectured that this process would:

- Motivate reviewers to find “hot spots” in the code that were especially vulnerable to small changes, thereby leading them to scrutinize places where malicious bugs were likely to have been inserted.
- Force reviewers to modify and run the program with the intention of producing a specific change in behaviour, thus requiring them to develop a deeper understanding of how the program works than they would get from merely reading the code.
- Yield a program with known bugs that could then be passed on to another group of reviewers to inspect. The existence of the known bugs would motivate the next group, and the fraction of those bugs they found could offer some measure of their effectiveness.

On the fourth day of the review, I asked the reviewers to try inserting their own bugs. Their experience led them to comment that being required to insert bugs might actually reduce a reviewer’s chances of finding bugs, because it would encourage reviewers to stick to the parts of code they already understand well, instead of diving deep into unfamiliar parts of the code.

What lessons were learned from the review?

Conducting software reviews.

- *Intentionally inserting bugs motivates reviewers.* The bug-insertion experiment created a dramatic difference in the review process. The reviewers became much more focused and motivated once they knew there was at least one bug to find, and the exercise became a lot more fun.
- *Set goals. Ask the reviewers specific questions, if you want answers.* Initially I assumed that the main outcome of the review would be an evaluation of the security and correctness of Pvote, and that the reviewers would arrive at some level of confidence that would raise or lower my level of confidence in Pvote's design and implementation. However, the review produced much broader discussion at many different levels: how to design programs to facilitate review, how to choose programming languages (or restricted subsets thereof) to facilitate review, and how to conduct reviews to maximize bug-finding effectiveness.
- *Static analysis, testing, and code review can make a good combination.* Each of these techniques alone has weaknesses: static analysis cannot enforce high-level requirements; testing cannot cover all possible inputs; and code review is tedious and error-prone. But in combination, they complement each other. Static analysis can reduce the tedium of code review by giving reviewers powerful starting assumptions. And testing—even cursory walkthroughs of the software—can quickly rule out flaws that break commonly used functionality. A bug that can get past both static analysis and live testing is a bug that causes trouble only in certain specific situations. It is likely to be nontrivial to write a bug that only causes misbehavior in specific situations, has a significant and intended effect on the outcome, and yet doesn't appear obviously unusual to a code reviewer.

Writing software to be reviewed.

- *Sometimes it is better to spell things out, even if it means more code.* Minimizing the number of lines of code was a high priority for me when I wrote the Pvote code. Although less code often means less work for reviewers, we discovered a few examples of the opposite. Minimizing complexity is not always the same as minimizing code.
- *The choice of language or language subset is important.* The language in which you write code heavily determines the amount of work that reviewers must do. The language design dictates the assumptions that reviewers are allowed to make. The choice of language also affects whether reviewers have tools to help them examine and analyze code more effectively.

Programming language design.

- *Supporting adversarial review is a new goal for programming languages.* Adversarial code review has demands that go beyond those of a typical code review. When the authors of the code are potentially malicious, they have a considerable home-turf advantage, as evidenced by the ability of an inserted bug to evade 20 reviewer-hours focused on just 100 lines of code.
- *Help programmers restrict parts of a program to subsets of the language.* Sometimes more language power is needed, sometimes less; sometimes different kinds of language features are needed for different purposes. Allowing the programmer to choose which subset of the language to use for each purpose can dramatically reduce the range of possible vulnerabilities that a reviewer has to consider.
- *Support for local reasoning is essential to adversarial review.* When reviewers are trying to verify a particular application-level property, they need ways to quickly rule out most of the program from being relevant to the assurance of that property. Any language feature that helps them perform local reasoning, or that lets the programmer create parts of the program where local reasoning is valid,

will make reviewing easier. Capability-style design is a promising approach, since it leverages lexical scope to support local reasoning [47].

Voting systems.

- *Pvote probably has fewer accidental bugs than most voting systems.* With 20 reviewer-hours focused on 100 lines (12 reviewer-minutes per line) and 90 reviewer-hours in total on the entire program, Pvote may be one of the most closely inspected pieces of voting software in existence, in terms of effort per line of code. (It would take ten person-years to review 100 000 lines of code with this much effort per line. Consider that most commercial voting systems contain hundreds of thousands of lines of code—in some cases over a million. Moreover, the complexity of code review probably increases more than linearly in the size of the code.) Since no bugs were found in the Pvote code, we can have some confidence that it meets a higher standard of code quality than the typical commercial voting system.
- *Detecting malicious code in a code review is extremely difficult.* Pvote was designed specifically to be minimal and written with code reviewing in mind. The reviewers had access to detailed documentation, as well as an environment that allowed them to modify and execute the program. Despite these things, and the high effort expended per line, an inserted bug went undetected. Though many of us expected that finding bugs would be difficult, we were still surprised by how hard it was.
- *Commercial voting systems are reviewed nowhere near enough to detect insider attacks.* Since the Pvote source code was probably reviewed more intensely than the source code of commercial voting systems has been reviewed, and since even this was insufficient to find a maliciously inserted bug, we can conclude that commercial voting systems almost certainly have not been subjected to the degree of review that would be necessary to declare it free of maliciously inserted bugs.

9 Complexity

Does prerendering actually eliminate complexity?	157
What is achieved by shifting complexity?	158
Why do software reviews assume trust in compilers?	160
How far back can the derivation of a program be traced?	161
What affects the tolerance of complexity in a component?	164
How does Pvote reallocate complexity?	167
What is gained by using interpreted languages?	173

Does prerendering actually eliminate complexity?

A theme running throughout this work is the management of complexity. The major unaddressed software threat is the insider threat from programmers; our only defense against it is assurance of software correctness. Complexity is the chief enemy of assurance, but it cannot be completely avoided. Prerendering the user interface is fundamentally a strategy for *mobilizing* complexity. The designer of the ballot definition language gains the freedom to move complexity that normally resides in the voting machine among three components:

- the tool that generates the ballot definition file,
- the ballot definition file, and
- the VM in the voting machine.

The allocation of complexity among these parts depends on design choices in the ballot definition language. For instance, in Pvote, the task of laying out buttons on the screen is no longer the job of the voting machine; it is in the ballot generation tool. The logic that decides when to play which audio message is no longer part of the voting machine; it is in the ballot definition.

Thus, prerendering does not, in itself, eliminate complexity; rather, it enables a designer to reallocate complexity. It is worthwhile to ask what this reallocation accomplishes. Does shifting complexity in this way make a real difference, or is it merely a shell game—a way of hiding complexity in components that I've conveniently chosen to ignore?

What is achieved by shifting complexity?

I argue that the reallocation of complexity does make a real difference. It matters where complexity resides because components differ in the way they are vulnerable, in the degree to which they are vulnerable, and in the people to whom they are vulnerable. Also, changing the allocation of complexity in a system has a significant effect because of the dependency relationships among the components.

To explain what I mean, I'll focus on just one of these relationships for a moment. The relationship I'm about to describe happens to be particularly important to the security of all software, not just voting machine software. When a software program runs, the instructions that the computer carries out are in an *executable file*. A *compiler* translates the source code into the executable file. The following figure depicts this relationship. The executable file is drawn as a larger box than the source code because it is usually larger and more complex. Typical compilers are enormously complex, so the compiler is the largest of all.

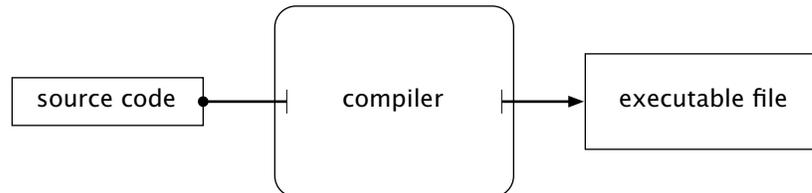


Figure 9.1. A compiler turns source code into an executable file. The sizes of the boxes (very roughly) indicate relative complexity.

When software undergoes a security review, the reviewers usually ask to look at the source code of the software, not the actual executable files. Source code is certainly *easier* to review than executable code. That's why programming languages were invented—so that humans would have something easier to deal with than low-level machine instructions. But convenience is not a reason for confidence.

If a thorough review of the executable file discovers no bugs, it directly offers (at least some) confidence that the executable file is correct. But if a thorough review of the source code discovers no bugs, it does not assure the correctness of the executable file unless the compiler is also correct.

Generative relationships like this exist throughout software systems. Whenever there is such a relationship, with an input, a transform, and an output, reviewers have a choice: they can inspect the output, or they can inspect the input and the transform instead. But it is necessary to establish that *both* the input and the transform are correct in order to establish that the output is correct.

In this example, the burden of establishing confidence in the executable is traded for the burden of establishing confidence in both the source code *and* the compiler. But a compiler is a massive piece of software—so why is this trade considered a good idea? In particular, why do software reviews typically skip inspection of the compiler? The next section looks at this question.

Why do software reviews assume trust in compilers?

Maybe they shouldn't. Not all computer scientists would agree that it is safe to assume a trustworthy compiler. In a famous essay on trust [76], Ken Thompson argued that compilers cannot be trusted, and gave a compelling demonstration of how to construct a deviously misbehaving compiler that would compile programs (including itself) incorrectly.

Despite Thompson's essay, much of current computer security practice (and even research) implicitly makes this assumption. One conceivable justification for this is that the compiler has a general purpose—it is designed to compile all sorts of programs—whereas the source code is written for a specific application. Perhaps those who trust compilers believe that the compiler is likely to be more mature and more thoroughly tested than a newly written program. Or perhaps they believe that, since the compiler is used to compile many different kinds of programs, someone would notice if it made compilation mistakes. Or perhaps—more depressingly—they simply think there is no hope of ever verifying compilers.

My purpose here is not to argue that corrupting a compiler in such a way would be impossible; clearly, as Thompson showed, it can be done. I aim only to offer *some* basis for the plausibility of the commonly held idea that corruption of a software program through subversion of the compiler is more difficult than directly corrupting the software's source code.

In choosing to review source code, reviewers trade an application-specific component with high complexity (the executable) for a component that is highly complex but general-purpose (the compiler), and a component that is application-specific but less complex (the source code).

How far back can the derivation of a program be traced?

What happens if you keep tracing where each component came from? The compiler is itself a piece of software; in Figure 9.1 it is shown as a mysterious box. What is that box, exactly? Is it the source code of the compiler or the executable file?

Actually, it is neither. The thing that actually performs the transformation of source code into an executable file is a *running instance* of the compiler. The transformation depicted by the “compiler” box is a process, not a static entity. So the following figure is a bit more accurate.



Figure 9.2. The middle box represents a compilation process, not a static piece of data.

The behaviour of that process is indeed derived from the executable file of the compiler program, but that is not all. Something has to turn that executable file (which is a static piece of data) into a running process; let us call this thing the *operating platform* on which it runs. The operating platform consists of all the software and hardware that makes it possible to run computer programs. It includes the operating system, software libraries, CPU, memory, storage, and so on—which makes it quite a bit bigger and more complex than the compiler.



Figure 9.3. An operating platform turns an executable file into a process.

The compiler executable was also derived from source code—the source code of the compiler—by an earlier compilation process. This earlier compilation may have been carried out by the same compiler or a different compiler. Putting all these relationships together gives us a fuller picture of how the executable program was derived.

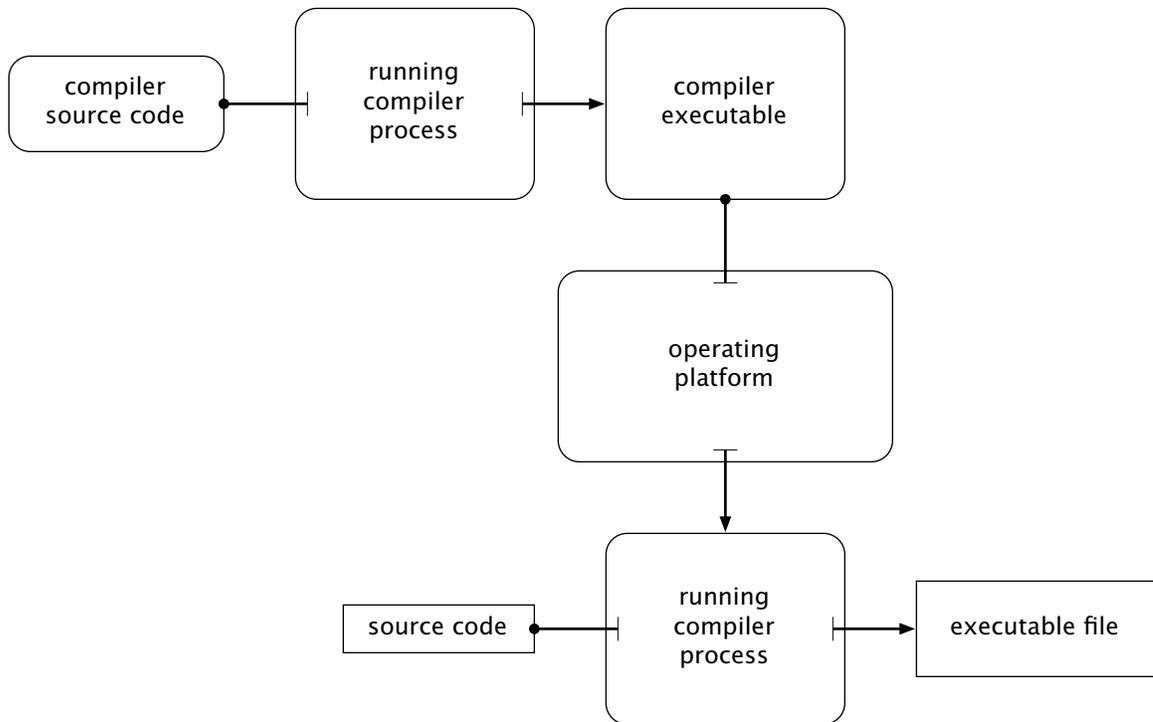


Figure 9.4. A small derivation map for a compiled program.

This diagram could continue indefinitely. The compiler process at the top of the diagram was itself produced by running a compiler executable on an operating platform, and *that* executable was the output of a compiler, and so on in a long chain of compilation steps running back through history. Ultimately the chain ends at an executable program that was created without the help of a compiler.

Malicious code that was introduced at any point in this chain could affect the final executable file. The program could be vulnerable to an insider attack that occurred many, many steps earlier—this is the point Thompson made in his essay.

There's still more to the picture—what about the operating platform? That, too, is constructed through a long chain of dependencies. It consists of operating system software compiled by a compiler, running on hardware produced by manufacturing processes that are also controlled by software.

I call these diagrams *derivation maps* because they show how a security-critical artifact is derived from other components. Each arrow represents a step in a hierarchical decomposition of the system. The purpose of this kind of analysis is to identify sources of vulnerability to insider attacks. Derivation maps can help you make an effective assurance argument or analyze an assurance argument to tell whether it is complete.

As a reviewer of the system, your challenge would be to cut away these sources of vulnerability. Each arrow in the diagram corresponds to a choice you could make: between reviewing the component at the head of the arrow and reviewing the two components at the tail and shaft of the arrow. Reviewing, testing, or otherwise establishing confidence in a particular component lets you ignore the arrowhead leading to it, and cut away the part of the diagram behind that arrowhead.

You may have noticed that some of the boxes in these diagrams have sharp corners and some have rounded corners. The reason for this is to indicate the distinction I mentioned earlier: general-purpose components have rounded corners, whereas application-specific components have sharp corners. This distinction is but one of many possible factors that could affect the degree to which one is willing to tolerate software complexity in a given component.

What affects the tolerance of complexity in a component?

Here are some of the ways in which you might evaluate a component with respect to the detectability of insider corruption. Classifying components according to these factors could help you identify ways that a shift in complexity can increase confidence.

- *User choice.* Are relying parties forced to use a particular implementation of the component, or do they have the freedom to choose their own? Shifting complexity from a dictated component to a freely chosen component reduces barriers to confidence. For example, anyone can choose or write their own tools to deconstruct and analyze ballot definition files. In contrast, voters cannot choose to vote on any equipment they want; they must use the equipment provided by election administrators.
- *Disclosure.* Is the component hidden or disclosed? The wider the audience to whom the component is disclosed, the harder it is for malicious code to go unnoticed. Components that are undisclosed, or inherently undisclosable (such as live running processes) are riskier because their correctness cannot be externally verified. Shifting complexity to a disclosed component reduces barriers to confidence.
- *Number of developers.* How many people have access to the component during development? If the component is authored by multiple people, corrupting it may require a conspiracy rather than just an individual attacker. Shifting complexity to a component with a larger development team might reduce barriers to confidence.
- *Specificity of purpose.* Shifting complexity from application-specific components to general-purpose components sometimes reduces barriers to confidence. Undetected bugs and backdoors may be less likely if the component is widely used and used in a variety of environments for a variety of purposes.

- *Testing.* Shifting complexity to components that have been thoroughly tested can reduce barriers to confidence, if the testing parallels the intended use.
- *Maturity.* How mature is the component? A component that has been stable, used, and developed for a long time has had more time to have its problems found and fixed. Shifting complexity to a more mature component could reduce barriers to confidence.
- *Release date.* When was the component released, relative to other components? Suppose, for example, that every time a particular compiler development team releases a new version of their compiler, the released version is reliably and indelibly archived. And suppose it can be verified that the compiler used to compile a particular program exactly matches the one released and archived on a particular date in the past. If the compiler was released before the program was even conceived, it is harder to imagine how an insider could have subverted the compiler to meaningfully influence the outcome of the program.
- *Reviewing resources.* There may be more reviewers or better reviewers available for certain types of components. For example, it might be easier to gain confidence in a component written in a more popular programming language because there is a larger community of people available who can understand and inspect the code.

Any of these factors could constitute a reason that shifting complexity from one component to another helps achieve better confidence.

While individual factors may not be enough to justify confidence, they can have stronger effects when combined. For example, even if a component has been tested thoroughly, there is still the possibility that it was written specifically to evade testing. But such evasion is likely to require some suspicious-looking code, which is less likely to escape notice if the code also happens to be disclosed to the public.

For a concrete example, consider Pvote. Suppose that Pvote will be run on a voting machine using version 2.3.5 of the Python interpreter, which was released in February 2005, before I started my research work on electronic voting. Python 2.3.5 is a mature open-source implementation of the language: it passes an extensive suite of functional tests, it has been widely used all over the world, and hundreds of programmers have contributed to its development.

Pvote's source code is also open to the public. If it were to be used for a real election, chances are good that it would be downloaded and examined by many people. Python is a well-known programming language with a large community of users who would be able to understand the Pvote code.

Given this context, how trustworthy is the Python interpreter? There are two ways that misbehaviour of the Python interpreter could be used in an insider attack:

- The Pvote program could be crafted to take advantage of a latent bug in the interpreter. The interpreter bug would have to be one that is not commonly triggered, since it would have survived years of open-source development and testing, as well as use with all kinds of Python programs. Yet at the same time, the Pvote code that triggers this unusual bug would also have to avoid looking out of the ordinary to the many Python programmers who inspect Pvote.
- The interpreter could be crafted to misbehave when running Pvote. To avoid detection in other contexts, the interpreter bug would have to be specific to the Pvote code in some way. But someone would have had to plant this bug in the interpreter before Pvote was designed and developed. The more specific the bug is to Pvote, the harder it is to see how the attacker could have predicted Pvote's implementation.

When it comes to software bugs, nothing is 100% certain. But when many positive factors come together in a context like this, they can constitute a basis for trust.

How does Pvote reallocate complexity?

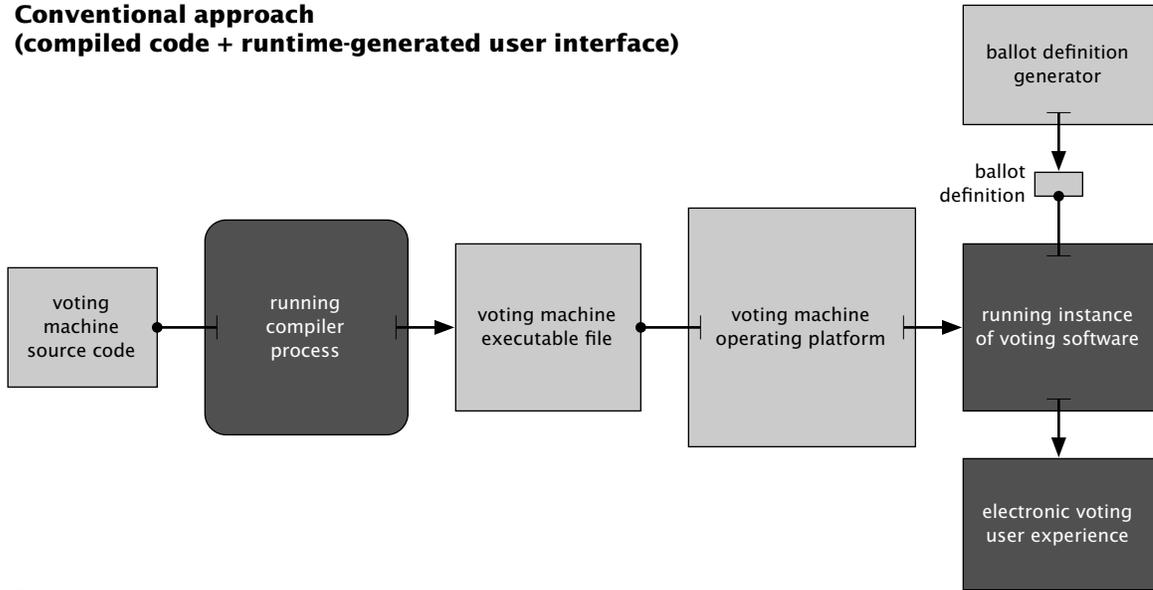
Figure 9.5 shows a derivation map for Pvote together with a derivation map of a conventional electronic voting machine for comparison. The ultimate product in each case is the *user experience* of the voter using the voting machine, which is determined by the voting machine software's interpretation of the ballot definition file. Both derivation maps omit the derivation of the compiler and operating platform.

Although the relative differences in size in the diagram are meant to roughly express relative differences in complexity, they are not to scale. For example, there is actually about 100 times as much source code in conventional voting machine software than there is in Pvote. Pvote is 460 lines of Python, whereas the Diebold AccuVote-TSx and the Sequoia Edge (two widely used touchscreen machines) run software consisting of 66 000 and 124 000 lines of code respectively [12, 7]. The complexity of a C compiler is many times larger still.

When you compare the two derivation maps, the two main complexity shifts are evident:

- *Ballot definition.* In Pvote, the ballot definition is more complex and the running instance of the voting VM is less complex than its counterpart in a conventional system, the running instance of the voting software. Also, the ballot definition is publicly disclosed.
- *Python interpreter.* In Pvote, the voting software runs on a Python interpreter rather than directly on the voting machine's operating platform. The source code to the voting VM is much smaller than that of the voting software in a conventional system; on the other hand, Pvote introduces the Python interpreter, a large additional component. Whereas the source code and executable for the voting machine software in a conventional system are application-specific and secret, the source code and executable for the Python interpreter used by Pvote are general-purpose and publicly disclosed.

**Conventional approach
(compiled code + runtime-generated user interface)**



**Pvote approach
(interpreted code + prerendered user interface)**

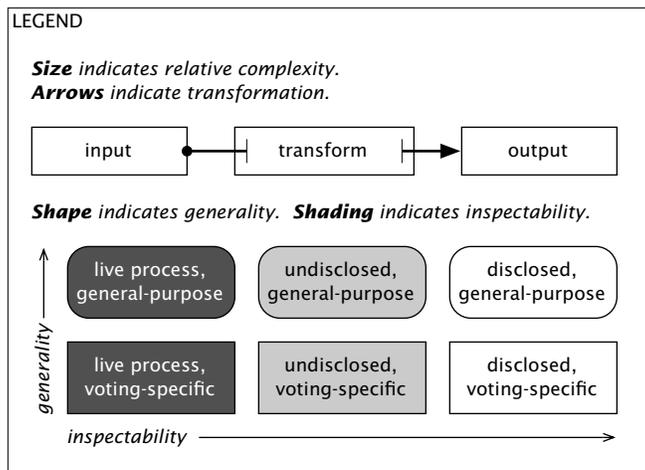
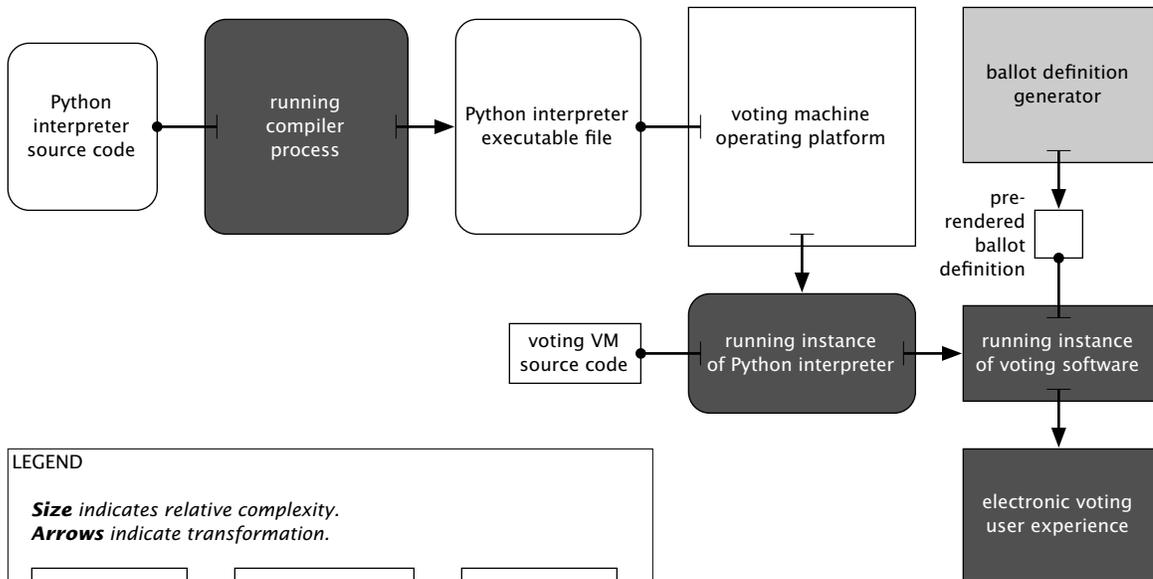


Figure 9.5. Derivation maps of a conventional voting system and of Pvote.

The effect of these architectural changes is to reduce the complexity of the *critical, voting-specific* components—the sharp-cornered boxes in the derivation map. Figure 9.5 highlights three factors about each component: complexity (size), generality of purpose (round or sharp corners), and disclosure (shading). In Pvote, the only voting-specific components that have to be inspected to gain confidence in the voting machine are the voting machine’s operating platform, the voting VM source code, and the prerendered ballot definition, and all three are disclosed.

Both changes are similar in character: in each case, a high-level interpreted language is introduced. Pvote replaces C with Python, and then replaces some of the Python code with a specialized ballot definition language. And in each case, the design of the high-level language dictates the balance of complexity between a pair of components in the diagram.

The following figure focuses on the relevant two pairs of components.

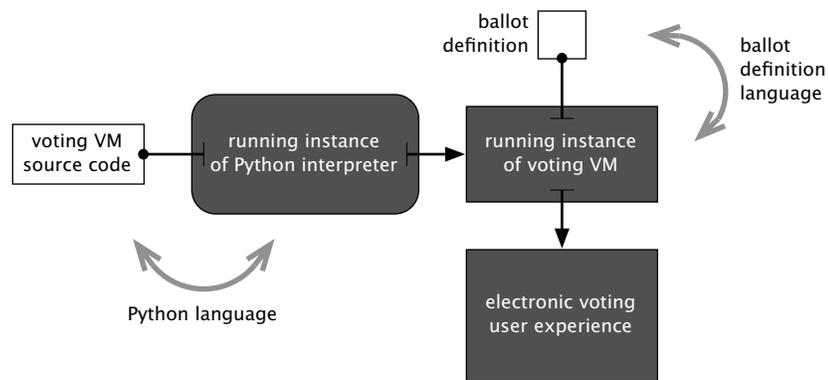


Figure 9.6. The two trade-offs introduced by Python and the ballot definition language.

The two boxes on the left trade off complexity according to how high-level the Python language is—that is, how much of the behaviour of the voting machine is specified by the Python interpreter as opposed to the source code it interprets. The diagrams on the next page explore what it would be like to move along the spectrum between using a low-level language and using a high-level language.

If the Python language were replaced with an extremely low-level language, the diagram would look like this:

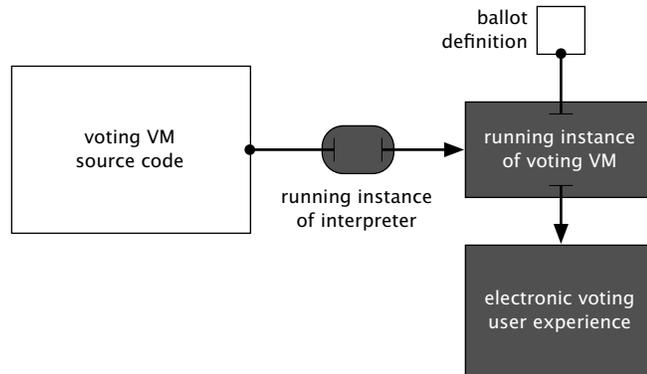


Figure 9.7. Python is replaced with a very low-level interpreter.

In the ultimate extreme, the interpreter would disappear and the input would no longer be source code; it would be an executable file running directly on the operating platform.

If the Python language were replaced with a higher-level language, the diagram would look like this:

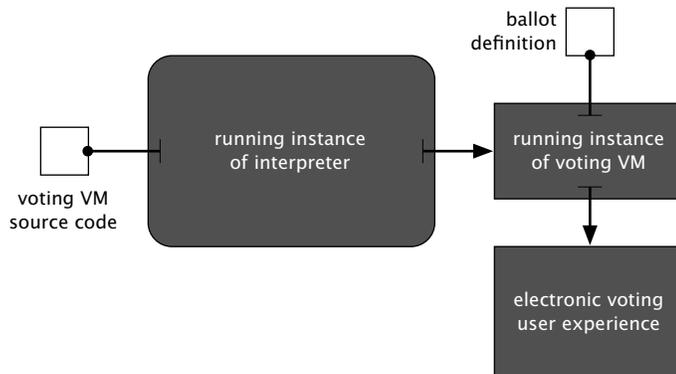


Figure 9.8. Python is replaced with a very high-level interpreter.

In the extreme, the input would disappear and the interpreter would subsume all the duties of the voting machine software—in effect, becoming the voting machine software.

The two extremes yield the same result: a specialized executable file running on the operating platform—exactly the situation of the conventional voting machine.

The two boxes at the top right trade off complexity according to the level of abstraction in the ballot definition language. With a very low-level ballot definition language, the diagram would look like this:

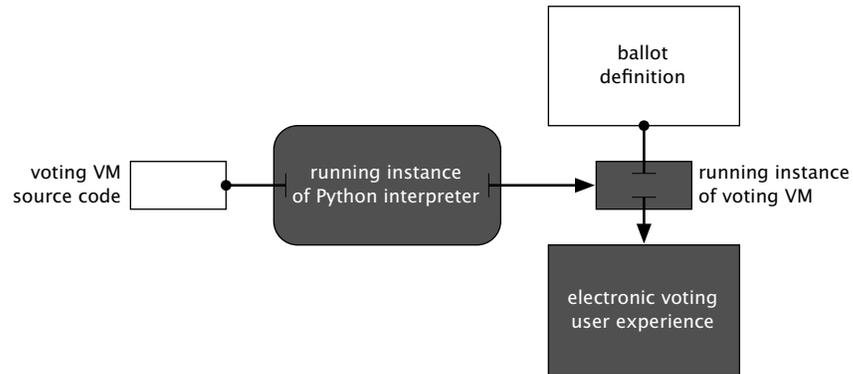


Figure 9.9. A low-level ballot definition language means a larger ballot definition.

In the extreme case, the VM would shrink to nothing at all, and the ballot definition would just be an executable file running on the voting machine.

With a very high-level ballot definition language, you get the following picture:

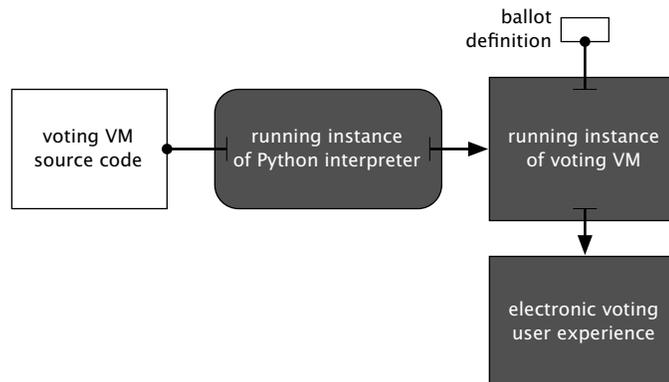


Figure 9.10. A high-level ballot definition language means a smaller ballot definition.

This is pretty much what happens in a conventional voting machine. Most of the voting user experience is defined by the voting machine software; the ballot definition only contains minimal information about the contests and candidates.

The conventional voting machine approach is about as far as it's possible to go in the direction of a high-level ballot definition language. That's because there has to be a way to configure the voting machine for the candidates and contests in a particular election; if we went any further, a specialized version of the voting machine software would have to be released for each ballot style.

Compared to conventional voting machine software, Pvote moves in the direction of a low-level ballot definition language. Giving the ballot definition language more power is beneficial because:

- it exposes more of the behaviour of the voting machine to public review,
- it exposes more of the behaviour of the voting machine to control by designers instead of programmers, and
- it allows the software in the voting machine to change less often. (Recall that back in Chapter 6, I said that greater generality in the ballot definition language helps to future-proof the voting VM software.)

But why not go so far as to shift all the complexity to the ballot definition, and eliminate the voting VM entirely? How do you choose the best balance between a high-level or low-level ballot definition, or between a high-level or low-level interpreted language for the voting machine software? The next section addresses these questions.

What is gained by using interpreted languages?

The purpose of programming language design is to offer high-level abstractions with which to express desired behaviour. The interpreter implements and enforces those abstractions. For example, the Python interpreter gives a guarantee of memory safety: in general, a Python program cannot arbitrarily corrupt memory. (There are extension modules designed specifically to allow arbitrary memory access, but the Python language definition excludes the use of such modules.) This both simplifies code written in Python and allows a reviewer of such code to make useful assumptions about its behaviour.

As another example, the ballot definition language contains no concept of the current time and date, and in general, no way to express behaviour that will be different at testing time than on election day itself. This property is essential to the effectiveness of “logic and accuracy testing,” in which behaviour observed in live pre-election testing is assumed to reflect the machine’s actual behaviour on election day. This restriction significantly reduces the amount of code that has to be reviewed to establish that the entire system has deterministic behaviour.

This is the answer to the question of balancing complexity between an interpreter and the code it interprets. Shifting complexity into a high-level programming language is useful only insofar as the target language provides security-relevant restrictions on what can be expressed. As long as a solid assurance argument can be made for the interpreter, it’s a good idea to make the interpreter responsible for abstractions that enforce useful correctness properties. In Python’s case, the argument is that Python is a general-purpose language; in the ballot definition language’s case, the argument is that the voting VM is small. My experience with Pvote suggests that restricted domain-specific languages and languages that support programming in restricted subsets are powerful tools for verifiable secure system design.

10 Related work

Do any other voting systems use prerendering?	175
What other voting proposals reduce reliance on software?	176
What are “frog” voting systems?	177
Do frogs solve the electronic voting problem?	178
What is “software independence” (SI)?	179
Does SI make software reliability irrelevant?	181
What is end-to-end (E2E) verification?	186
Does E2E verification make software reliability irrelevant?	187
What are other approaches to high-assurance software?	188

Do any other voting systems use prerendering?

Yes, there is some precedent for using prerendered images in electronic voting machines.

The Open Voting Consortium's EVM2003 project [59, 58] used a full-screen bitmap image for displaying an electronic ballot.¹ This use of a prerendered image was also motivated by a desire for software simplicity.

The ES&S iVotronic supports the use of "bitmap ballots" for displaying ballots in foreign languages [36].² These ballots contain graphical images for the candidate's names and other text, so that text in arbitrary languages can be shown.

To the best of my knowledge, Pvote is the first voting system that uses a prepared description of the entire user interface, including full-screen images, prerecorded audio, and a specification of behaviour. This extension of the concept of prerendering is significant for all the reasons identified in Chapter 4: it further simplifies the software in the voting computer, enables more thorough public review, creates a more complete public record, gives designers control over ballot design, and reduces the need to change the voting computer software.

¹According to David Mertz of the OVC, this idea was originally proposed for use in EVM2003 by Fred McLain.

²My thanks are due to Dan Wallach for mentioning this precedent to me.

What other voting proposals reduce reliance on software?

Many voting researchers have recognized the difficulty of testing and verifying software, and sought to reduce the vulnerability of elections to software bugs or maliciously crafted software. The prerendering approach is motivated by the desire to reduce the size and complexity of the trusted base on which the security of the voting system rests. In the following sections, I'll discuss other major proposals that share the same motivation:

- The “frog” voting scheme
- “Software independence” (and a common implementation of SI, the voter-verified paper audit trail)
- End-to-end verification schemes

What are “frog” voting systems?

In 2001, researchers from CalTech and MIT proposed a voting procedure based on “frogs” [10]. They coined the term “frog” to mean a small and cheap device, such as a memory card, that permanently stores a single voter’s votes—the electronic equivalent of an individual marked paper ballot.

The frog proposal separates the voting process into two steps, vote selection and vote casting, each carried out with a separate machine. The voter first selects their votes on the vote-selection machine, which stores them on a frog. The voter then puts the frog into the vote-casting machine, which displays the contents of the frog for the voter to check, and upon confirmation by the voter, casts the votes. The frog is kept as a permanent record in case a recount is needed later.

The idea behind this proposal is to separate the more complicated operation of selecting votes from the security-sensitive operation of casting the votes. According to the proposers, the trusted base of software is reduced because responsibility for security now rests only on the simpler vote-casting machine; the vote-selection machine will have “no need for high security” [10].

Do frogs solve the electronic voting problem?

Not entirely. The central claim of the frog scheme—that it excludes the vote-selection software from the trusted base—relies on two significant assumptions:

- that voters will check their frogs carefully before casting them, and
- that voters will know what to expect when the contents of the frog are displayed.

Some voters may give the vote-casting machine only a cursory glance, and most are likely to be influenced by confirmation bias [55]. Thus, it is possible—perhaps even likely—that votes recorded incorrectly by the vote-selection machine could go unnoticed. The susceptibility of an election to incorrect recording by the vote-selection machine also depends on how election administrators respond when voters report problems, and how many complaints are needed to trigger such response.

Even if voters do check the votes on their frogs carefully, the vote-selection machine remains in a position to influence voters during the selection process—thus violating the principle that an election should be an unbiased measurement. For example, the vote-selection machine could present the candidates in a biased way. It could change the wording of a ballot measure to make an option seem more appealing or even invert the sense of the question, swapping the implications of “yes” and “no”. It could even give misleading instructions to voters, such as telling them to ignore the vote-casting machine or to go to a different polling place to vote on certain contests.

The prerendered approach therefore targets a broader security goal: to secure the entire voting user interface including the vote selection process, in order to avoid bias in the election’s measurement of the will of the electorate. Prerendering the user interface does not rule out the possibility of further partitioning the user interface into two steps as proposed in the frog voting architecture.

What is “software independence” (SI)?

“Software independence” is a prominent concept in the next version of U. S. federal standards for voting systems, the “2007 VVSG.” A draft of the 2007 VVSG [81] has been unanimously adopted by the standards committee, but remains open for public comment before adoption. Section 2.4 of that draft introduces the term like this:

Software independence means that an undetected error or fault in the voting system’s software is not capable of causing an undetectable change in election results.

The draft declares that “All voting systems must be software independent to conform to the VVSG.” The draft goes on to explain the concept like this:

There are essentially two issues behind the concept of software independence, one being that it must be possible to audit voting systems to verify that ballots are being recorded correctly, and the second being that testing software is so difficult that audits of voting system correctness cannot rely on the software itself being correct.

According to the draft:

- Hand-counted paper ballots and optically scanned paper ballots are software independent, since they leave a paper record that can later be recounted by hand to check that the original counts are correct.
- DRE machines with a VVPAT feature are also software independent, since the VVPAT records are on paper and can also be recounted by hand.
- DRE machines without paper trails are not software independent (even though some DREs offer a “recount” function, this is carried out by just another software program and so fails to be software independent).

The name and concept of “software independence” were introduced in a white paper by Rivest and Wack [66] written for the committee that was working on the VVSG. In addition to giving a definition of “software independence” (essentially the same as the one quoted above), this paper identified a distinction between “strong software-independence” and “weak software-independence.” A strongly software-independent voting system is one for which changes in outcome due to software errors are not only detectable but also correctable without re-running the election. A weakly software-independent voting system is one that has the detection property (i.e., satisfies the above definition of “software independence”) without a recovery mechanism. Essentially, “strong software independence” is “software independence” plus a recovery mechanism.

Does SI make software reliability irrelevant?

No. Requiring all voting systems to provide a software-independent audit capability is certainly an important improvement, but this alone is far from what would be necessary to achieve confidence in a voting system.

To explain why, I need to go into a bit of detail about how the term “software independence” is used in the VVSG draft. The VVSG draft defines the term with one meaning and then uses it with a second meaning—and unfortunately, neither of these two meanings actually constitute independence from software. There are three main problems with the VVSG definition and the use of the name “software independence” for the concept:

1. The VVSG definition does not describe systems that are actually independent of software, just systems that are *less than totally* dependent on software.
2. The meaning of the VVSG definition depends on detection procedures that are unspecified.
3. The use of the term in the VVSG focuses on auditing the counting of recorded votes, but elections can be influenced in many ways other than miscounting or altering recorded votes.

Less-than-total dependence is not independence. The initial definition of “software independence” given in Section 2.4 of the VVSG draft requires that software faults be “not capable of causing an undetectable change” in the election outcome. If the software can cause an undetectable change, then the election is 100% reliant upon the software to be correct. But as long as any software-caused change is *detectable in principle*, no matter how vanishingly small the probability of detection, the voting system will meet the definition. Even a voting system that has only a 0.1% chance of error detection (and is thus, in a sense, 99.9% dependent on software) would meet the VVSG definition of “software independent.”

The detection procedures are unspecified. By using the word “undetectable,” the VVSG definition presumes the existence of some procedures by which errors could be detected. However, it does not specify whether those procedures need to be realistic or practical.

For example, the VVSG draft says that DRE machines without paper trails fail to be “software independent.” Consider for the sake of argument a DRE machine with no VVPAT that stores vote records on a cassette tape (as old microcomputers like the TRS-80 and Apple II used to do). In principle one could stop the machine and examine the electronic records after each ballot is cast, thereby detecting incorrectly recorded votes; this examination would require some electronic equipment but could be performed without software. Does such a DRE machine therefore meet the definition, despite lacking a paper trail?

As another example, consider a DRE machine that produces a paper audit trail with the vote information printed as a barcode. Is it “software independent”? If recounts of the paper audit trail are performed using a barcode scanner, then the recount would depend on the software that processes the barcodes. Yet, in principle, a human being with enough patience could examine the stripes in the barcode, decode them by hand, and thus conduct a software-independent audit. Whether this machine meets the definition of “software independence” depends on assumptions about what one uses to perform the detection.

Further, what constitutes successful detection? In some analyses of the probability of software fault detection, detection by a single voter constitutes detection. But a complaint from a single voter is unlikely to stop an election, cause machines to be taken out of service, or launch an investigation. This is for good reason: if election administrators made it their policy to take any machine out of service based on a complaint from a single voter, just a few dishonest voters could effectively shut down polling stations and cause havoc on election day. Thus election officials must choose some threshold of voter complaints they deem necessary to trigger remedial action.

How should the proper threshold be determined? If the threshold is too low, the election will be vulnerable to fraudulent complaints. If the threshold is too high, the election will be vulnerable to undetected faults. It may even be the case that there is no acceptable threshold of voter complaints because these two ranges of unacceptable thresholds overlap. The likelihood of recovery from a software fault is intimately dependent on the policies for response and escalation when problems are reported.

It should be clear from the preceding analysis that software independence is necessarily a property of an entire election administration system, including policies and procedures as well as technology. I propose the following definition:

***True software independence (TSI)** means there is a negligible probability that an error or fault in the voting system's software will change the outcome of the election.*

For clarity, I will use "VSI" to refer to the VVSG definition:

***VVSG software independence (VSI)** means an undetected error or fault in the voting system's software cannot cause an undetectable change in the outcome of the election.*

Although the definitions are similar, the difference between "a negligible probability of change" and "no undetectable change" is significant. The first describes something that can be estimated and measured; the second does not, and depends on unstated assumptions about what is detectable, what detection procedures are performed, and what constitutes successful detection.

"Strong software independence" (SSI) as defined by Rivest and Wack [66] and TSI are both stronger versions of the VSI concept, but they strengthen the concept in different ways. SSI adds recovery to VSI, but a voting system can still meet SSI even if the probability of detection and recovery is minimal. TSI requires that the probability of detection and recovery be high.

Altering recorded votes is not the only way to influence an election. Immediately after presenting the VSI definition, the VVSG draft then explains the term “software independence” with a different meaning: namely, the capability to audit the counting of votes without relying on software. Here is the relevant excerpt from the VVSG draft (emphasis added):

*There are essentially two issues behind the concept of software independence, one being that **it must be possible to audit voting systems to verify that ballots are being recorded correctly**, and the second being that testing software is so difficult that **audits of voting system correctness cannot rely on the software itself being correct**. ... [P]revious versions [of the VVSG] permitted voting systems that are software dependent, that is, voting systems whose audits must rely on the correctness of the software.*

I will use the term “software-independent audit capability” to refer to this concept:

*A voting system has **software-independent audit capability (SIAC)** if it provides a procedure for verifying that votes were recorded and counted correctly without relying on the correctness of any software.*

SIAC has a narrower meaning than VSI, because it is only concerned with the counting of votes after they are recorded. Faulty voting machines can influence elections in many other ways—for example, by presenting the candidates in a biased fashion, omitting contests from the ballot, misleading the voter with false instructions, printing incorrect paper audit trails, or crashing and preventing voters from casting votes at all.

A DRE with a voter-verified paper audit trail (VVPAT) can influence an election in all of these ways, and so it fails to be TSI even though it has SIAC. All of these are ways that an election would, in fact, depend on software, despite being called “software independent” according to the VVSG draft.

It can even be argued that a DRE with a VVPAT fails to meet the VSI definition, depending on the interpretation of the word “undetectable.” Consider, for example, a DRE with a VVPAT, which is programmed to occasionally skip a particular contest on the first time through the ballot. The contest is only skipped the first time through, and the contest is still printed on the VVPAT as usual.

Imagine the typical voter’s experience with this machine. After going through all the pages of the ballot, the voter might or might not read the VVPAT carefully. The VVPAT will show that no selection was made for the skipped contest; the voter has no way to tell whether the software maliciously skipped the contest, the voter missed a page due to double-tapping on the “next page” button by mistake, or the voter just forgot to fill in that contest. In any case, if the voter goes back and fills in the missing vote, everything behaves normally.

A malicious DRE such as this can exert significant influence on an election. Yet it leaves no evidence that would show that the software is at fault; that is, no amount of forensic analysis after the election would be able to establish that a contest was unfairly skipped. The emphasis on auditing in the VVSG draft’s use of the term “software independence” suggests that recorded evidence is centrally important. If “undetectable” in the VSI definition means “not detectable by examination of recorded evidence,” then DREs with VVPATs fail to be VSI.

If DREs with VVPATs *are* VSI, it seems strange to define “software independent” such that machines with software in a position to mislead voters qualify as “software independent.”

Why software reliability still matters. Even if a voting system qualifies as SIAC or even VSI according to the definitions I’ve identified here, there are still many ways that the election can be vulnerable to software faults—for example, crashing more frequently for voters of a particular political party. If software presents the ballot to the voter, then software is in a position to mislead or otherwise influence the voter. Therefore, software reliability and correctness remain vital to election integrity.

What is end-to-end (E2E) verification?

As mentioned in Chapter 3, “end-to-end verification” is the name for a family of techniques that enable each individual voter to verify that his or her votes were properly counted in the final total. The main challenge of end-to-end verification is to provide enough information for voters to perform this check, yet not enough information for voters to sell their votes.

The general approach of E2E schemes is to publish a complete but anonymous record of all the votes so that anyone can check the count; where the schemes differ is in how they assure voters that their individual votes are included in the published record of votes.

- Some schemes publish a set of encrypted, identifiable vote records in addition to the complete set of plaintext, anonymous vote records. These include VoteHere [54], Scratch & Vote [1], Prêt-à-Voter [13], and Punchscan [26]. Voters receive an encrypted record of their votes to take home, which they can check against a published encrypted record. Some other mathematical procedure is used to verify that the two sets of vote records correspond.
- Some schemes give each voter a record with only partial information about his or her votes to take home. The information is enough to check against the published records but insufficient as sellable evidence of his or her votes. ThreeBallot [68] and VAV [67] fall into this category.
- Twin [67] is an unusual end-to-end scheme. In Twin, each voter receives a receipt for a randomly selected other voter’s ballot. Thus, while the posted records can be matched with receipts, they can’t be identified as belonging to any particular voter.

Does E2E verification make software reliability irrelevant?

End-to-end verification schemes let voters ensure their votes are counted without relying on software. Voters using an E2E voting system have all the information they need to perform this check themselves—unlike voters using a voting system with a VVPAT, who must rely on election administrators to conduct a hand count of the VVPATs in order for the paper record to matter. Thus, E2E schemes provide the potential for stronger voter verifiability, as long as voters are willing to carry out a more involved procedure to verify their votes.

However, E2E schemes do not address the problems of ballot presentation and crashing software. Purely paper-based E2E schemes avoid the use of computers for vote entry, but may limit access for voters with some kinds of disabilities. On the other hand, if the ballot is presented by a computer or votes are entered on a computer, the problems of reliable ballot presentation and vote entry remain; it is these issues that prerendering addresses. Programs like Pvote can provide the reliable vote-entry functionality needed for computer-based E2E voting systems.

What are other approaches to high-assurance software?

Automated proof. The desire to prove software programs to be correct has existed pretty much since programmable computers were invented. As early as 1961, John von Neumann sought to mathematically prove the correctness of computer programs [30]. Since that time, researchers have investigated a variety of ways to automatically construct a proof that a program meets a formal specification.

- *Verification conditions.* In 1969, James King developed an automatic program verifier [42] based on associating verification conditions with execution paths through the program. Each verification condition is the proposition that if an initial predicate (i.e., a precondition) holds at the beginning of the execution path, then a final predicate (i.e., a postcondition) will hold when the end of the execution path is reached. The correctness of the entire program is established by proving that all these verification conditions hold, and showing that their paths can be chained together to cover all possible execution paths from where the program starts to where the program halts.

A modern example of this approach is Java Modelling Language (JML). Programmers can embed JML annotations in comments in Java code to specify assertions such as invariants, preconditions, and postconditions. A static checking tool called ESC/Java [27] can then analyze the program and verify the consistency of these assertions.

- *Weakest precondition methods.* The weakest precondition approach works in the opposite direction. It begins with the desired postcondition and works backwards through the program to determine the weakest precondition that would be necessary to imply the postcondition.
- *Abstract interpretation.* Abstract interpretation [16] (also known as symbolic execution) consists of executing the statements of a program using an abstract representation of

the program's state. That is, instead of giving concrete values to variables, an abstract interpreter keeps track of an expression representing each variable's value in terms of the input. These expressions evolve as variables are manipulated, and may take on a disjunction of the values produced by conditional branching. Proofs of properties about these expressions are then used to establish the correctness of the program.

- *Model checking.* In the model checking approach, software engineers must first construct a model of their program design or requirements in a formal modelling language. Then an automatic prover checks that the model meets a set of desired properties, which also have to be specified in a formal notation.

Each of the above techniques has to rely on an automated theorem prover to show that symbolic logical statements about the program imply the desired properties to be verified. One of the earliest theorem provers used for checking programs was the Boyer-Moore theorem prover, also known as NQTHM. A review article by Boyer and Moore [8] reports that NQTHM has been used to check large systems such as a microprocessor design, an assembler, and a small operating system kernel. ACL2 [39], the successor to NQTHM, is one of the best known modern theorem provers. Simplify [19] is another well-known automatic theorem prover that serves as the proving engine for ESC/Java.

The prerendering technique does not compete with these formal approaches; instead, it augments their power. All of the above methods require a formal specification against which to check the program and, in the case of model checking, a formal model of the program itself. A formally verified program is only as correct as the specification against which it was verified. Creating such specifications and models correctly is a tricky task. A smaller and simpler original program makes the specifications, models, and resulting proofs less likely to contain mistakes.

During the Pvote security review, we discussed the possibility of translating Pvote into a language where there is support for formal verification, and adding the necessary annotations for preconditions and postconditions. The two main options we talked about were Java (which has JML and ESC/Java) and SPARK Ada [6], a commercially developed variant of Ada specifically designed for high assurance and verification.

Proof-carrying code. In the proof-carrying code (PCC) technique [53], the supplier of an application constructs a formal proof that it satisfies a security policy, and includes this proof (in encoded form) in the distributed application binary. The host system on which the application will be run can then check the proof for itself, without relying on any other trusted parties, to ensure that the program is safe to run.

In the context of electronic voting, the PCC approach would require the voting machine to run a proof checker. PCC proof checkers have been built as small as 2 700 lines [4] (about 30% of which are in C and the rest in Twelf, a logic specification language), but this is still substantially larger than Pvote.

Formal code generation. Instead of applying machine analysis to check the correctness of human-written code, an alternative is to machine-generate code in such a way that the code must be correct. This is the concept behind formal code generation [86]. A human-written specification still has to direct the machine generation of code, but this specification could be written at a higher level, in a declarative rather than a procedural manner.

Large-scale program analysis. Several tools have been developed for analyzing large programs for bugs. These tools make no attempt to prove correctness; they are mainly intended to catch specific kinds of common errors that the programmer may have missed. A recent example of such a project is Oink (based on CQual++ [28]), which has been used to scan the Debian Linux codebase for format string vulnerabilities [14].

Conclusion

In this dissertation, I've examined the problem of electronic voting, starting from an analysis of the requirements for a democratic election and the different kinds of voting systems used in practice and proposed by researchers. This analysis led me to focus on the correctness and simplicity of the software in the voting computer, a challenge I've addressed through the technique of user interface prerendering. This concept led to two iterations of design and implementation, culminating in the creation of Pvote, a vote-entry program that supports synchronized audio and video, touchscreen input, and accessible device input.

Pvote is implemented in just 460 lines of Python—a tiny amount of code compared to existing voting machines such as the Diebold AccuVote-TSx (66 000 lines of code) or the Sequoia Edge (124 000 lines of code)—yet it allows a high degree of flexibility in the design of the user interface. With Pvote, the user interfaces of voting computers can finally be designed by experts in information design, interaction design, and accessibility instead of voting system programmers. The security review of Pvote's design and source code is reason for optimism about Pvote's correctness. Although the results showed that Pvote was not reviewed enough to be positive that it lacks flaws, the review also found no bugs in Pvote despite intense scrutiny. Pvote validates the prerendered user interface approach by demonstrating that it can meet both accessibility and security goals.

The quest to create reliable voting machine software has yielded some results that can be applied to high-assurance software of other kinds. This work focused specifically on defending against the insider attack, a long-standing and

difficult problem in computer security that has rarely been addressed. User interface prerendering is an effective technique whenever a general-purpose computer is used for a specialized purpose and high reliability is required despite periodic changes in the user interface. Derivation maps are helpful for analyzing and mitigating potential sources of vulnerability to insider attacks. The experience with the Pvote security review yielded insights into language and design features that would support the adversarial code review process, and redoubled my respect for how difficult it can be to review code written by a potential adversary. The review experience has convinced me that small teams and short timeframes are inadequate for adversarial review, and suggests that true confidence in voting system software is likely to require source code disclosure to the public or a large community of reviewers, for an extended period of time before use in an election.

Will we ever create electronic voting machines are truly worthy of trusting with our votes? I can't predict whether we will, but at least one thing is established: Pvote puts a stake in the ground to show just how small voting machine software can be. There is simply no good reason to rely on voting machine software that's hundreds of times larger.



Bibliography

- [1] Ben Adida and Ronald L. Rivest (2006). Scratch & vote: self-contained paper-based cryptographic voting. In *Proceedings of the 5th ACM Workshop on Privacy in the Electronic Society*, pages 29–40. ACM Press.
- [2] Alan Agresti and Brett Presnell (2002). Misvotes, Undervotes and Overvotes: The 2000 Presidential Election in Florida. *Statistical Science*, 17(4):436–440 (Voting and Elections, November 2002). Institute of Mathematical Statistics.
- [3] Edward G. Amoroso (1994). *Fundamentals of Computer Security Technology*. Prentice Hall.
- [4] Andrew W. Appel, Neophytos G. Michael, Aaron Stump, and Roberto Virga (2002). A Trustworthy Proof Checker. Technical Report TR-648-02, Department of Computer Science, Princeton University. Available at <http://www.cs.princeton.edu/research/techreps/TR-648-02>.
- [5] Jonathan Bannet, David W. Price, Algis Rudys, Justin Singer, and Dan S. Wallach (2004). Hack-a-Vote: Security Issues with Electronic Voting Systems. *IEEE Security & Privacy*, 2(1):32–37 (January/February 2004).
- [6] John Barnes (1997). *High Integrity Ada: The SPARK Approach*. Addison-Wesley.
- [7] Matt Blaze, Arel Cordero, Sophie Engle, Chris Karlof, Naveen Sastry, Micah Sherr, Till Stegers, and Ka-Ping Yee (2007). *Source Code Review of the Sequoia Voting System*. Available at http://www.sos.ca.gov/elections/voting_systems/ttbr/sequoia-source-public-jul26.pdf.
- [8] Robert S. Boyer and J. Strother Moore (1990). A Theorem Prover for a Computational Logic. *Lecture Notes in Computer Science*, (449):1–15 (July 1990).

- [9] Steven J. Brams and Peter C. Fishburn (1998). Approval Voting. *The American Political Science Review*, 72(3):831–847 (September 1998).
- [10] Shuki Bruck, David Jefferson, and Ronald L. Rivest (2001). A Modular Voting Architecture (“Frogs”). Presented at the Workshop on Trustworthy Elections (WOTE 2001). Available at <http://www.vote.caltech.edu/wote01/pdfs/amva.pdf> (retrieved on June 7, 2007).
- [11] Darren Burton and Mark Uslan (2002). Cast a Vote by Yourself: A Review of Accessible Voting Machines. *AccessWorld*, November 2002. Available at <http://www.afb.org/afbpress/pub.asp?docid=aw030603>.
- [12] Joseph A. Calandrino, Ariel J. Feldman, J. Alex Halderman, David A. Wagner, Harlan Yu, and William P. Zeller (2007). *Source Code Review of the Diebold Voting System*. Available at http://www.sos.ca.gov/elections/voting_systems/ttbr/diebold-source-public-jul29.pdf.
- [13] David Chaum, Peter Ryan, and Steve A. Schneider (2004). A Practical, Voter-verifiable Election Scheme. Technical Report CS-TR-880, School of Computing Science, University of Newcastle upon Tyne, UK.
- [14] Karl Chen and David A. Wagner (2007). Large-Scale Analysis of Format String Vulnerabilities in Debian Linux. In *Proceedings of the 2007 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS 2007)*, pages 75–84. ACM Press.
- [15] Lillie Coney (2004). Statement Before the U. S. Election Assistance Commission, Technical Guidelines Development Committee, September 22, 2004. Available at http://vote.nist.gov/voting_statement.pdf (retrieved on December 4, 2007).
- [16] Patrick Cousot and Radhia Cousot (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Conference on Principles of Programming Languages*, pages 238–252. ACM Press.
- [17] Theo de Raadt. OpenBSD Security. Available at

- <http://www.openbsd.org/security.html> (retrieved on December 13, 2007).
- [18] Stephanie Delaune, Steve Kremer, and Mark Ryan (2006). Coercion-resistance and Receipt-freeness in Electronic Voting. In *Proceedings of the 19th Computer Security Foundations Workshop (CSFW)*. IEEE Computer Society Press.
- [19] David Detlefs, Greg Nelson, and James B. Saxe (2003). Simplify: A Theorem Prover for Program Checking. Technical Report 2003-148, Hewlett-Packard Labs. Available at <http://www.hp1.hp.com/techreports/2003/HPL-2003-148.html>.
- [20] Diebold Election Systems. Welcome to Diebold Election Systems. <http://www.dieboldes.com/> as of January 24, 2004. Archived copy available at <http://web.archive.org/web/20040209133249/www2.diebold.com/dieboldes/default.htm>.
- [21] Christopher Drew (2007). U. S. Bars Lab From Testing Electronic Voting. *New York Times*, January 4, 2007.
- [22] Election Data Services (2004). Overview of Voting Equipment Usage in United States, Direct Recording Electronic (DRE) Voting. Statement of Kimball Brace to the United States Election Assistance Commission, May 5, 2004. Available at http://www.electiondataservices.com/EDSInc_DREoverview.pdf.
- [23] Election Reform Information Project (2007). Voter-Verified Paper Audit Trail Legislation & Information. Available at <http://www.electionline.org/Default.aspx?tabid=290> (retrieved on December 13, 2007).
- [24] Sarah P. Everett, Michael D. Byrne, and Kristen K. Greene (2006). Measuring the usability of paper ballots: Efficiency, effectiveness, and satisfaction. In *Proceedings of the Human Factors and Ergonomics Society 50th Annual Meeting*. Human Factors and Ergonomics Society.
- [25] Sarah P. Everett (2007). *The Usability of Electronic Voting Machines and How Votes Can Be Changed Without Detection*. Ph. D. dissertation, Department of Psychology, Rice University. Available at <http://chil.rice.edu/alumni/petersos/EverettDissertation.pdf>.

- [26] Kevin Fisher, Richard Carback, and Alan T. Sherman (2006). Punchscan: Introduction and System Definition of a High-Integrity Election System. In *Proceedings of the IAVoSS Workshop On Trustworthy Elections (WOTE 2006)*.
- [27] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata (2002). Extended static checking for Java. *ACM SIGPLAN Notices*, 37(5):234-245 (May 2002).
- [28] Jeffrey S. Foster (2002). *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. Ph. D. dissertation, Computer Science Division, University of California, Berkeley.
- [29] Laurin Frisina, Michael C. Herron, James Honaker, and Jeffrey B. Lewis (2007). Ballot Formats, Touchscreens, and Undervotes: A Study of the 2006 Midterm Elections in Florida. Available at <http://www.dartmouth.edu/~herron/cd13.pdf> (retrieved on June 7, 2007).
- [30] H. Goldstine and John Von Neumann (1961). Planning and Coding Problems. In A. H. Taub, editor, *John Von Neumann Collected Works*, number V, pages 80-152. Pergamon Press.
- [31] Rop Gonggrijp and Willem-Jan Hengeveld (2007). Studying the Nedap/Groenendaal ES3B Voting Computer: A Computer Security Perspective. In *Proceedings of the USENIX/ACCURATE Electronic Voting Technology Workshop (EVT 2007)*. USENIX Press.
- [32] Bev Harris (2004). *Black Box Voting: Ballot Tampering in the 21st Century*. Talion Publishing. Available at <http://www.blackboxvoting.org/book.html>.
- [33] Harri Hursti (2006). Diebold TSx Evaluation: Critical Security Issues with Diebold TSx, May 11, 2006. Available at <http://www.blackboxvoting.org/BBVtsxstudy.pdf> (retrieved on June 7, 2007).
- [34] Information Technology Association of America, Election Technology Council (2006). Comments submitted on behalf of the ITAA ETC to the members of the State of California Senate Committee on Elections, Reapportionment and

Constitutional Amendments, February 8, 2006. Archived at <http://web.archive.org/web/20060622084803/http://www.electiontech.org/downloads/ITAA+ETC+CA+OSS+TESTIMONY+-+FINAL.pdf>.

- [35] Srinivas Inguva, Eric Rescorla, Hovav Shacham, and Dan S. Wallach (2007). *Source Code Review of the Hart InterCivic Voting System*. Available at http://www.sos.ca.gov/elections/voting_systems/ttbr/Hart-source-public.pdf.

- [36] Douglas W. Jones (2004). Recommendations for the Conduct of Elections in Miami-Dade County using the ES&S iVotronic System. Available at <http://www.cs.uiowa.edu/~jones/voting/miami.pdf> (retrieved on December 13, 2007).

- [37] Douglas W. Jones (2006). Connecting Work on Threat Analysis to the Real World. Presented at Threat Analyses for Voting System Categories: A Workshop on Rating Voting Methods (VSRW 2006). Available at <http://www.cs.uiowa.edu/~jones/voting/VSRW06.pdf> (retrieved on December 19, 2007).

- [38] Ari Juels, Dario Catalano, and Markus Jakobsson (2005). Coercion-Resistant Electronic Elections. In *Proceedings of the 2005 ACM Workshop on Privacy in the Electronic Society*, pages '61-70. ACM Press.

- [39] Matt Kaufmann and J. Strother Moore (1996). ACL2: An Industrial Strength Version of Nqthm. In *Proceedings of the Eleventh Annual Conference on Computer Assurance, Systems Integrity, Software Safety, and Process Security*, pages 23-34.

- [40] Arthur Keller (2007). Experiences with Sequoia AVC Edge with VeriVote Printer as Precinct Inspector in Santa Clara County.

- [41] Tim P. Kelly (1998). *Arguing Safety - A Systematic Approach to Managing Safety Cases*. Ph. D. dissertation, Department of Computer Science, University of York. Available at <http://www-users.cs.york.ac.uk/~tpk/tpkthesis.pdf>.

- [42] James C. King (1969). *A program verifier*. Ph. D. dissertation, Department of Computer Science, Carnegie Mellon University.

- [43] Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach (2004). Analysis of an Electronic Voting System. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press.
- [44] Samuel Merrill (1988). *Making Multicandidate Elections More Democratic*. Princeton University Press.
- [45] Adrian Mettler and David A. Wagner. Joe-E (open source software project). Available at <http://joe-e.org/>.
- [46] Joanne M. Miller and Jon A. Krosnick (1998). The Impact of Candidate Name Order on Election Outcomes. *Public Opinion Quarterly*, 62(3):291-330.
- [47] Mark S. Miller (2006). *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph. D. dissertation, Department of Computer Science, Johns Hopkins University. Available at <http://erights.org/talks/thesis>.
- [48] David Molnar, Tadayoshi Kohno, Naveen Sastry, and David A. Wagner (2006). Tamper-Evident, History-Independent, Subliminal-Free Data Structures on PROM Storage -or- How to Store Ballots on a Voting Machine. In *Proceedings of the 2006 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press.
- [49] Moni Naor and Vanessa Teague (2001). Anti-persistence: History Independent Data Structures. In *Proceedings of the 33rd Annual ACM Symposium on the Theory of Computing*, pages 492-501. ACM Press.
- [50] National Conference of State Legislatures. Straight Ticket Voting States (2007). Available at http://www.ncsl.org/programs/legismgt/elect/straight_ticket.htm (retrieved on December 13, 2007).
- [51] National Council on Disability (2006). NCD Statement: Voluntary Voting System Guidelines. Available at http://www.ncd.gov/newsroom/publications/2006/voluntary_voting.htm (retrieved on December 13, 2007).

- [52] National Institute of Standards and Technology (1995). FIPS 180-1: Secure Hash Standard. Available at <http://www.itl.nist.gov/fipspubs/fip180-1.htm> (retrieved on December 13, 2007).
- [53] George C. Necula (1997). Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1997)*, pages 106–119. ACM Press.
- [54] C. Andrew Neff and Jim Adler (2003). Verifiable e-voting. Available at http://votehere.com/vhti/documentation/VH_VHTi_WhitePaper.pdf (retrieved on December 13, 2007).
- [55] Raymond S. Nickerson (1998). Confirmation Bias: A Ubiquitous Phenomenon in Many Guises. *Review of General Psychology*, 2(2):175–220.
- [56] Richard G. Niemi and NIST (2005). Sample State and Local Ballots. Available at <http://vote.nist.gov/ballots.htm> (retrieved on December 13, 2007).
- [57] Nokia Corporation. Python for Series 60 (open source software project). Available at <http://opensource.nokia.com/projects/pythonfors60>.
- [58] Open Voting Consortium. EVM2003 (open source software project). Available at <http://evm2003.sourceforge.net/>.
- [59] Open Voting Consortium (2007). Ballot Prerendering. Available at <http://www.openvotingconsortium.org/ballot-prerendering.html> (retrieved on December 18, 2007).
- [60] Thea Peacock and Peter Ryan (2006). Coercion-resistance as Opacity in Voting Systems. Technical Report CS-TR-959, School of Computing Science, University of Newcastle upon Tyne, UK.
- [61] Elliot Proebstel, Sean Riddle, Francis Hsu, Justin Cummins, Freddie Oakley, Tom Stanionis, and Matt Bishop (2007). An Analysis of the Hart Intercivic DAU eSlate. In *Proceedings of the USENIX/ACCURATE Electronic Voting Technology Workshop (EVT 2007)*. USENIX Press.

- [62] Pygame (open source software project). Available at <http://pygame.org/>.
- [63] Python Software Foundation. Python (programming language). Available at <http://www.python.org/>.
- [64] RABA Technologies (2004). Trusted Agent Report: Diebold AccuVote-TS Voting System, January 20, 2004. Available at http://www.raba.com/press/TA_Report_AccuVote.pdf (retrieved on December 19, 2007).
- [65] ReportLab, Inc. ReportLab Toolkit (open source software project). Available at <http://www.reportlab.org/>.
- [66] Ronald L. Rivest and John P. Wack (2006). On the notion of “software independence” in voting systems. Available at <http://vote.nist.gov/SI-in-voting.pdf> (retrieved on December 5, 2007).
- [67] Ronald L. Rivest and Warren D. Smith (2007). Three Voting Protocols: ThreeBallot, VAV, and Twin. In *Proceedings of the USENIX/ACCURATE Electronic Voting Technology Workshop (EVT 2007)*. USENIX Press.
- [68] Ronald L. Rivest (2006). The ThreeBallot Voting System. Available at <http://people.csail.mit.edu/rivest/Rivest-TheThreeBallotVotingSystem.pdf> (retrieved on June 7, 2007).
- [69] Noel H. Runyan (2007). Improving Access to Voting: A Report on the Technology for Accessible Voting Systems. Dēmos and Voter Action. Available at http://demos.org/pubs/improving_access.pdf (retrieved on June 7, 2007).
- [70] Bruce Schneier (1999). Attack Trees: Modeling security threats. *Dr. Dobb's Journal*, 24(12):21-29 (December 1999).
- [71] Markus Schulze (2003). A New Monotonic and Clone-Independent Single-Winner Election Method. *Voting Matters*, (17):9-19 (October 2003).
- [72] Science Applications International Corporation (2003). Risk Assessment Report: Diebold AccuVote-TS Voting System and Processes, September 2, 2003. Available at

http://www.elections.state.md.us/pdf/risk_assessment_report.pdf
(retrieved on November 14, 2007).

- [73] Ted Selker (2005). Voting Technology: Election Auditing is an End-to-End Procedure. *Science*, 308(5730):1873-1874 (June 2005).
- [74] Warren D. Smith (2000). Range Voting. Available at <http://math.temple.edu/~wds/homepage/rangevote.pdf> (retrieved on December 13, 2007).
- [75] Molly F. Story (1998). Maximizing Usability: The Principles of Universal Design. *Assistive Technology*, 10(1):4-12.
- [76] Ken Thompson (1984). Reflections on Trusting Trust. *Communications of the ACM*, 27(8):761-763 (August 1984).
- [77] T. Nicolaus Tideman (1987). Independence of Clones as a Criterion for Voting Rules. *Social Choice and Welfare*, 4(3):185-206 (September 1987).
- [78] United States: 107th Congress (2002). Help America Vote Act of 2002. Available at http://www.fec.gov/hava/law_ext.txt.
- [79] United States: 110th Congress (2007). H. R. 811: Voter Confidence and Increased Accessibility Act of 2007. Available at <http://thomas.loc.gov/cgi-bin/bdquery/z?d110:h.r.00811:>.
- [80] United States Election Assistance Commission (2005). *2005 Voluntary Voting System Guidelines*. Available at <http://www.eac.gov/voting%20systems/voluntary-voting-guidelines/2005-vvsg>.
- [81] United States Election Assistance Commission (2007). *Draft 2007 Voluntary Voting System Guidelines*. Available at <http://www.eac.gov/files/vvsg/Final-TGDC-VVSG-08312007.pdf> (retrieved on November 17, 2007).
- [82] Verified Voting Foundation (2004). Election 2004 E-Voting Incidents from the Election Incident Reporting System. Available at

<http://www.verifiedvotingfoundation.org/article.php?id=5331>.

- [83] William E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl (1981). *Fault Tree Handbook (NUREG-0492)*. United States Nuclear Regulatory Commission.
- [84] David A. Wagner, David Jefferson, Matt Bishop, Chris Karlof, and Naveen Sastry (2006). Security Analysis of the Diebold AccuBasic Interpreter, February 14, 2006. Available at <http://www.cs.berkeley.edu/~daw/papers/accubasic.pdf> (retrieved on June 7, 2007).
- [85] Jonathan N. Wand, Kenneth W. Shotts, Jasjeet S. Sekhon, Jr. Walter R. Mebane, Michael C. Herron, and Henry E. Brady (2001). The Butterfly Did It: The Aberrant Vote for Buchanan in Palm Beach County, Florida. *American Political Science Review*, 95(4):793–810 (December 2001).
- [86] Michael W. Whalen and Mats P. E. Heimdahl (1999). On the Requirements of High Integrity Code Generation. In *Proceedings of the 4th IEEE International Symposium on High-Assurance Systems Engineering*. IEEE Computer Society Press.
- [87] S. P. Wilson, Tim P. Kelly, and John A. McDermid (1996). Safety Case Development: Current Practice, Future Prospects. In *Safety and Reliability of Software-Based Systems*, page 135. Springer-Verlag New York.
- [88] Alec Yasinsac, David A. Wagner, Matt Bishop, Ted Baker, Breno de Medeiros, Gary Tyson, Michael Shamos, and Mike Burmester (2007). *Software Review and Security Analysis of the ES&S iVotronic 8.0.1.2 Voting Machine Firmware*. Florida Department of State.
- [89] Ka-Ping Yee and Mark Miller (2002). Auditors: An Extensible, Dynamic Code Verification Mechanism. Available at <http://www.erights.org/elang/kernel/auditors/> (retrieved on December 13, 2007).
- [90] Ka-Ping Yee (2006). Prerendered User Interfaces for Higher-Assurance Electronic Voting. In *Proceedings of the USENIX/ACCURATE Electronic Voting Technology Workshop (EVT 2006)*. USENIX Press.

- [91] Ka-Ping Yee (2007). Extending prerendered-interface voting software to support accessibility and other ballot features. In *Proceedings of the USENIX/ACCURATE Electronic Voting Technology Workshop (EVT 2007)*. USENIX Press.
- [92] Ka-Ping Yee (2007). Pvote Software Review Assurance Document. Technical Report EECS-2007-40, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, CA. Available at <http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-40.html>.
- [93] Ka-Ping Yee (2007). Report on the Pvote security review. Technical Report EECS-2007-136, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley. Available at <http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-136.html>.

A Ptouch source code

The following pages present the source code of Ptouch, consisting of five modules:

- `main.py`
- `Ballot.py`
- `Navigator.py`
- `Video.py`
- `Recorder.py`

Each line of code is numbered and printed in monospaced type.

```
36 flags = [0 for c in m.contests]
```

Defining occurrences of classes, methods, and functions appear in bold.

```
123 def getlist(ballot, stream, Class):
```

Lines marked with a triangle are entry points into a module, called from other modules. Functions and methods without a triangle are called only from within the same module.

```
▷ 45 def activate(self, slot_i):
```

The code is broken into sections, with explanatory text in grey preceding each section.

Explanatory text looks like this.

main.py

This is the main Ptouch program. It initializes the other software components with the provided ballot definition file and then processes incoming Pygame events in a non-terminating loop.

```
1 import Ballot, Navigator, Recorder, Video
2 from pygame import display, event, MOUSEBUTTONDOWN, KEYDOWN
```

The following lines load and verify the ballot definition, then instantiate the other parts of Ptouch with their corresponding sections of the ballot definition.

```
3 ballot = Ballot.Ballot('ballot')
4 video = Video.Video(ballot.imagelib)
5 recorder = Recorder.Recorder(ballot)
6 navigator = Navigator.Navigator(ballot.model, video, recorder)
```

This is the main event loop. The loop begins by updating the display to match the framebuffer in memory, so that any display changes made during the last iteration appear onscreen. The loop never exits.

```
7 while 1:
8     display.update()
```

On each iteration, one event is retrieved from Pygame's event queue. The only type of event Ptouch handles is a mouse click. The coordinates of the mouse click are translated into a slot index. If the click corresponds to a slot, it is passed to the navigator's `activate()` method for further handling.

```
9     e = event.wait()
10    if e.type == MOUSEBUTTONDOWN:
11        slot = video.locate(*e.pos)
12        if slot is not None:
13            navigator.activate(slot)
```

Ballot.py

The `Ballot` module defines the ballot definition data structure. The main program instantiates a `Ballot` object to deserialize the ballot data from a file stream and construct the ballot definition data structure. All the other classes in this module represent parts of the ballot definition; each one deserializes its contents from the stream passed to its constructor.

```
1 class Ballot:
▷ 2     def __init__(self, filename):
3         self.data = open(filename).read()
4         stream = open(filename)
```

`sprite_n` is a counter that keeps track of the next sprite index. Each instance of the `Option`, `Writein`, `Subpage`, and `Subtarget` classes contains a local field called `sprite_i` that points to its associated sprite. This field is set by the `__init__` method of the class, which picks up the sprite index by accessing and incrementing the `sprite_n` field of the `Ballot` during loading. `subpage_n` is a local counter of subpages that is only used during verification after the ballot is loaded.

```
5         self.sprite_n = subpage_n = 0
6         self.model = m = Model(self, stream)
7         self.imagelib = il = ImageLib(self, stream)
8         assert stream.read(1) == ''
```

At this point the ballot definition has been fully loaded into memory. The rest of the `__init__` method verifies that the ballot definition is well-formed. If it is not well-formed, the program should be aborted with a fatal error to prevent the possibility that Ptouch will crash after starting a voting session.

The following lines ensure that there is at least one page and one contest, and that the arrays of layouts and sprites have the proper sizes.

```
9         assert m.pages and m.contests
10        assert len(m.pages) + len(m.subpages) == len(il.layouts)
11        assert len(il.sprites) == self.sprite_n
```

`items` contains one list corresponding to each contest; it will collect all the slots and sprites for the options in the contest. `chars` also contains one line corresponding to each contest; it will collect all the slots and sprites for the write-in characters in the contest. These lists will later be checked to ensure that the sizes of all sprites match the sizes of the slots into which they could be pasted.

```
12        items = [[] for c in m.contests]
13        chars = [[] for c in m.contests]
```

For each page, the targets, options, write-ins, and reviews are checked to ensure their fields have valid values.

```
14        for i, p in enumerate(m.pages):
15            for t in p.targets:
16                assert t.action in [0, 1, 2]
17                assert 0 <= t.page_i < len(m.pages)
18            for x in p.targets + p.options + p.writeins + p.reviews:
19                assert 0 <= x.contest_i < len(m.contests)
```

The `slot` variable keeps track of the slot index during checking of the slots associated with each page.

```
20         slots = il.layouts[i].slots
21         slot = len(p.targets)
```

The slots and sprites for all the option areas are gathered into the appropriate arrays for later size checking.

```
22         for i, o in enumerate(p.options):
23             items[o.contest_i] += [slots[slot + i], il.sprites[o.sprite_i]]
```

The slots and sprites for all the write-ins are gathered into the appropriate arrays for later size checking.

```
24         slot += len(p.options)
25         for w in p.writeins:
26             items[w.contest_i] += [slots[slot], il.sprites[w.sprite_i]]
27             max_chars = m.contests[w.contest_i].max_chars
28             chars[w.contest_i] += slots[slot + 1:slot + 1 + max_chars]
29             slot += 1 + max_chars
```

The slots and sprites for all the review areas are gathered into the appropriate arrays for later size checking.

```
30         for r in p.reviews:
31             max_chars = m.contests[r.contest_i].max_chars
32             for i in range(m.contests[r.contest_i].max_sels):
33                 items[r.contest_i] += [slots[slot]]
34                 chars[r.contest_i] += slots[slot + 1:slot + 1 + max_chars]
35                 slot += 1 + max_chars
```

The `flags` array indicates which contests contain write-in options.

```
36         flags = [0 for c in m.contests]
37         for p in m.pages:
38             for w in p.writeins:
39                 flags[w.contest_i] = 1
```

For each contest with write-in options, the associated write-in subpage is checked to ensure it has the right number of slots and all of its subtargets have fields with valid values. The slots for write-in characters are gathered into the appropriate arrays for later size checking. In this loop, `subpage_n` keeps track of the index of the associated subpage.

```
40         for i, c in enumerate(m.contests):
41             if flags[i]:
42                 c.subpage_i, subpage_n = subpage_n, subpage_n + 1
43                 p = m.subpages[c.subpage_i]
44                 slots = il.layouts[len(m.pages) + c.subpage_i].slots
45                 assert len(p.subtargets) + c.max_chars == len(slots)
46                 chars[i] += slots[len(p.subtargets):]
47                 for t in p.subtargets:
48                     assert t.action in [0, 1, 2, 3, 4, 5]
49                     if t.action in [0, 1]:
50                         chars[i] += [il.sprites[t.sprite_i]]
51                 chars[i] += [il.sprites[p.cursor_i]]
```

The number of subpages in the ballot model should match the number of contests with write-in options, which were counted in the preceding loop.

```
52         assert len(m.subpages) == subpage_n
```

Each layout is checked to ensure that its background image matches the screen size and all its slots are positioned within the screen bounds.

```
53     for l, b in [(l, l.background) for l in il.layouts]:
54         assert (b.width, b.height) == (il.width, il.height)
55         for slot in l.slots:
56             assert 0 <= slot.left < slot.left + slot.width < il.width
57             assert 0 <= slot.top < slot.top + slot.height < il.height
```

Finally, the sprites and slots that have been collected for each group are checked to ensure they all have properly matching sizes.

```
58     for list in items + chars:
59         for x in list:
60             assert (x.width, x.height) == (list[0].width, list[0].height)
```

Each remaining class loads its contents from the stream in a constructor that parallels its data structure. These constructors instantiate other classes to read single components from the stream, call `getlist()` to read a variable-length list of components from the stream, or call `getint()` to deserialize an integer from the stream.

```
61 class Model:
62     def __init__(self, ballot, stream):
63         self.contests = getlist(ballot, stream, Contest)
64         self.pages = getlist(ballot, stream, Page)
65         self.subpages = getlist(ballot, stream, Subpage)
66
67 class Contest:
68     def __init__(self, ballot, stream):
69         self.max_sels = getint(stream)
70         self.max_chars = getint(stream)
71
72 class Page:
73     def __init__(self, ballot, stream):
74         self.targets = getlist(ballot, stream, Target)
75         self.options = getlist(ballot, stream, Option)
76         self.writeins = getlist(ballot, stream, Writein)
77         self.reviews = getlist(ballot, stream, Review)
78
79 class Target:
80     def __init__(self, ballot, stream):
81         self.action = getint(stream)
82         self.page_i = getint(stream)
83         self.contest_i = (self.action == 1 and [getint(stream)] or [0])[0]
84
85 class Option:
86     def __init__(self, ballot, stream):
87         self.contest_i = getint(stream)
88         self.sprite_i, ballot.sprite_n = ballot.sprite_n, ballot.sprite_n + 1
89
90 class Writein:
91     def __init__(self, ballot, stream):
92         self.contest_i = getint(stream)
93         self.sprite_i, ballot.sprite_n = ballot.sprite_n, ballot.sprite_n + 1
94
95 class Review:
96     def __init__(self, ballot, stream):
97         self.contest_i = getint(stream)
```

```

92 class Subpage:
93     def __init__(self, ballot, stream):
94         self.subtargets = getlist(ballot, stream, Subtarget)
95         self.cursor_i, ballot.sprite_n = ballot.sprite_n, ballot.sprite_n + 1
96
97 class Subtarget:
98     def __init__(self, ballot, stream):
99         self.action = getint(stream)
100        if self.action in [0, 1]:
101            self.sprite_i, ballot.sprite_n = ballot.sprite_n, ballot.sprite_n + 1
102
103 class Imageib:
104     def __init__(self, ballot, stream):
105         self.width = getint(stream)
106         self.height = getint(stream)
107         self.layouts = getlist(ballot, stream, Layout)
108         self.sprites = getlist(ballot, stream, Image)
109
110 class Layout:
111     def __init__(self, ballot, stream):
112         self.background = Image(ballot, stream)
113         self.slots = getlist(ballot, stream, Slot)
114
115 class Slot:
116     def __init__(self, ballot, stream):
117         self.left = getint(stream)
118         self.top = getint(stream)
119         self.width = getint(stream)
120         self.height = getint(stream)

```

An **Image** object contains the pixel data for an image, which resides in a single Python string. In serialized form, the image's width and height are stored preceding the pixel data, which contains three bytes per pixel (one byte each for the red, green, and blue components).

```

118 class Image:
119     def __init__(self, ballot, stream):
120         self.width = getint(stream)
121         self.height = getint(stream)
122         self.pixels = stream.read(self.width * self.height * 3)

```

The **getlist()** function reads a variable-length list of data structures from the stream, all of a particular given class. In Python (and Pthin), classes are first-class objects and can be passed as arguments. In serialized form, the list is preceded by a 4-byte integer indicating how many elements to read.

```

123 def getlist(ballot, stream, Class):
124     return [Class(ballot, stream) for i in range(getint(stream))]

```

The **getint()** function reads an unsigned 4-byte integer from the stream, serialized with the most significant byte first.

```

125 def getint(stream):
126     bytes = [ord(char) for char in stream.read(4)]
127     return (bytes[0]<<24) + (bytes[1]<<16) + (bytes[2]<<8) + bytes[3]

```

Navigator.py

The navigator is initialized with access to the ballot model data structure, the video driver, and the vote recording module. It saves these references locally, initializes an empty selection state, and begins the voting session by transitioning to page 0. The `selections` member contains a list of selections for each contest. The elements of these lists are themselves lists: an ordinary selected option is represented by a list of a single integer, the option's sprite index; a selected write-in option is represented by a list containing the write-in option's sprite index followed by the indices of the character sprites entered for the write-in.

```
1 class Navigator:
▷ 2     def __init__(self, model, video, recorder):
3         self.model, self.video, self.recorder = model, video, recorder
4         self.selections = [[] for contest in model.contests]
5         self.goto(0)
6         self.update()
```

The `goto()` method transitions to a given page. If the transition goes to the last page, the voter's selections are recorded. Any page transition clears the `writein` and `chars` members, which are set only when a subpage is active (`writein` points to the current write-in object, and `chars` contains the write-in characters entered so far).

```
7     def goto(self, page_i):
8         if page_i == len(self.model.pages) - 1:
9             self.recorder.write(self.selections)
10            self.page_i, self.page = page_i, self.model.pages[page_i]
11            self.writein, self.chars = None, []
```

The `update()` method updates the video display based on the current page and selections.

```
12     def update(self):
```

When the `writein` member is not `None`, this means the user is currently on a subpage. The video driver is told to paste the subpage's background over the entire screen, then paste any entered characters into the character slots of the subpage, in order. If the character slots are not all full, the cursor sprite is also pasted into the next available character slot.

```
13         if self.writein:
14             contest = self.model.contests[self.writein.contest_i]
15             subpage = self.model.subpages[contest.subpage_i]
16             self.video.goto(len(self.model.pages) + contest.subpage_i)
17             offset = len(subpage.subtargets)
18             for i, sprite_i in enumerate(self.chars):
19                 self.video.paste(sprite_i, offset + i)
20             if len(self.chars) < contest.max_chars:
21                 self.video.paste(subpage.cursor_i, offset + len(self.chars))
```

When the `writein` member is `None`, no subpage is active. The video driver is told to paste the current page's background over the entire screen, then fill in the options, write-ins, and reviews on the page according to the current selections. The indices of the corresponding slots are assumed to be arranged in sequential order, as described in Chapter 5; hence the variable `slot_i` is incremented in each loop and carried forward to the next loop.

```

22         else:
23             self.video.goto(self.page_i)

```

To check whether an option is selected, the elements of the contest's selection list are scanned for a one-element list containing the option's sprite index.

```

24         slot_i = len(self.page.targets)
25         for option in self.page.options:
26             if [option.sprite_i] in self.selections[option.contest_i]:
27                 self.video.paste(option.sprite_i, slot_i)
28                 slot_i += 1

```

To check whether a write-in is selected, the elements of the contest's selection list are scanned for a list whose first element is the write-in option's sprite index. If such a list is found, the rest of the elements in the list are the sprite indices of the entered characters, so all the sprites in the list can be pasted into the write-in's slots in the order they appear. (The cursor is not shown on ordinary pages, only on subpages.)

```

29         for writein in self.page.writeins:
30             for selection in self.selections[writein.contest_i]:
31                 if selection[0] == writein.sprite_i:
32                     for j, sprite_i in enumerate(selection):
33                         self.video.paste(sprite_i, slot_i + j)
34                 slot_i += 1 + self.model.contests[writein.contest_i].max_chars

```

To display a review, the selections in the contest's selection list are pasted into the review's slots in the order they appear. Since write-in selections are represented by a list beginning with the write-in sprite index followed by the entered character sprites, these sprites will fit into the `1 + contest.max_chars` slots corresponding to the review. The inner loop always executes `contest.max_sels` times so that `slot_i` will be incremented by the correct amount.

```

35         for review in self.page.reviews:
36             contest = self.model.contests[review.contest_i]
37             selections = self.selections[review.contest_i]
38             for i in range(contest.max_sels):
39                 if i < len(selections):
40                     for j, sprite_i in enumerate(selections[i]):
41                         self.video.paste(sprite_i, slot_i + j)
42                 slot_i += 1 + contest.max_chars

```

The `activate()` method activates a slot when a user touches the touchscreen within the slot. The triggered behaviour depends on whether the slot corresponds to a subtarget, a target, an option, or a write-in.

```
▷ 43     def activate(self, slot_i):
```

When the `writein` member is not `None`, this means the user is currently on a subpage. The touched slot index is treated as a subtarget index. The `action` field of the subtarget determines the action to take: the values from 0 through 5 correspond to `APPEND`, `APPEND2`, `DELETE`, `CLEAR`, `CANCEL`, and `ACCEPT`.

```
44         if self.writein:
45             contest = self.model.contests[self.writein.contest_i]
46             subpage = self.model.subpages[contest.subpage_i]
47             subtarget = subpage.subtargets[slot_i]
```

`APPEND` appends the selected character. `APPEND2` appends the selected character only if the write-in is not empty. In both cases the character is only appended if the maximum length will not be exceeded.

```
48             if subtarget.action == 0 or subtarget.action == 1 and self.chars:
49                 if len(self.chars) < contest.max_chars:
50                     self.chars += [subtarget.sprite_i]
```

`DELETE` deletes the last entered character.

```
51             if subtarget.action == 2:
52                 self.chars[-1:] = []
```

`CLEAR` clears all the entered characters.

```
53             if subtarget.action == 3:
54                 self.chars = []
```

`CANCEL` cancels the write-in and exits the subpage. The write-in option was already removed from the selection list upon entry to the subpage (see line 85), so upon return to the original page, the write-in option will be cleared and deselected.

```
55             if subtarget.action == 4:
56                 self.goto(self.page_i)
```

`ACCEPT` accepts the write-in and exits the subpage. The write-in sprite and entered character sprites are placed into a list, and this list is added to the selection list for this contest.

```
57             if subtarget.action == 5 and self.chars:
58                 self.selections[self.writein.contest_i] += [
59                     [self.writein.sprite_i] + self.chars]
60                 self.goto(self.page_i)
```

The rest of the cases cover user actions when the user is on an ordinary page. The first case covers targets; the `action` field of the target can be 0, 1, or 2, corresponding to a plain transition, a transition with clearing the selections in a contest, and a transition with clearing all the selections in the entire ballot.

```

61         elif slot_i < len(self.page.targets):
62             target = self.page.targets[slot_i]
63             if target.action == 1:
64                 self.selections[target.contest_i] = []
65             if target.action == 2:
66                 self.selections = [[] for contest in self.model.contests]
67             self.goto(target.page_i)

```

The next case handles options. Touching an option toggles whether it is selected, unless this would exceed the selection limit indicated by the contest's `max_sels` field.

```

68         elif slot_i < len(self.page.targets) + len(self.page.options):
69             option = self.page.options[slot_i - len(self.page.targets)]
70             selections = self.selections[option.contest_i]
71             contest = self.model.contests[option.contest_i]
72             if [option.sprite_i] in selections:
73                 selections.remove([option.sprite_i])
74             elif len(selections) < contest.max_sels:
75                 selections += [[option.sprite_i]]

```

The only remaining case is that the user has touched a write-in. In this case, `slot_i` is used to find the appropriate write-in, and its contest's selection list is searched to see whether the write-in is already selected.

```

76         else:
77             slot_i -= len(self.page.targets) + len(self.page.options)
78             for writein in self.page.writeins:
79                 contest = self.model.contests[writein.contest_i]
80                 if slot_i < 1 + contest.max_chars:
81                     selections = self.selections[writein.contest_i]
82                     for i, selection in enumerate(selections):

```

If the write-in is already selected, the write-in characters that were previously entered need to be moved into the `chars` buffer so they will appear on the subpage. The entry for this write-in in the selection list is removed upon entry to the subpage; it will be added back if the user decides to accept the write-in (see line 58).

```

83                     if selection[0] == writein.sprite_i:
84                         self.writein, self.chars = writein, selection[1:]
85                         selections[i:i + 1] = []
86                         break
87

```

If the write-in is not selected, its subpage is simply activated.

```

88                 else:
89                     if len(selections) < contest.max_sels:
90                         self.writein = writein
91                     break
92                 slot_i -= 1 + contest.max_chars

```

The display is then updated to reflect the selection changes and/or transition that were enacted in response to the user's touch.

```

93             self.update()

```

Video.py

Video display control is provided by the pygame library.

```
1 from pygame import display, image, FULLSCREEN
```

The `loadimage()` function converts a string containing uncompressed pixel data into a Pygame `Image` object.

```
2 def loadimage(i):
3     return image.fromstring(i.pixels, (i.width, i.height), 'RGB')
```

The `Video` class is responsible for pasting full-screen images and sprites onto the display, as well as translating touch locations into slot indices.

```
4 class Video:
```

The video driver is initialized with access to the image library section of the ballot definition. It initializes the Pygame display and converts all the images from raw data into Pygame `Image` objects.

```
> 5     def __init__(self, il):
6         display.init()
7         self.screen = display.set_mode((il.width, il.height), FULLSCREEN)
8         self.backgrounds = [loadimage(l.background) for l in il.layouts]
9         self.layouts = [l.slots for l in il.layouts]
10        self.sprites = [loadimage(sprite) for sprite in il.sprites]
11        self.goto(0)
```

The `goto()` method switches to a given layout, which involves pasting the layout's background image over the entire screen. The `slots` member always points to the current layout's slots.

```
> 12     def goto(self, layout_i):
13         self.slots = self.layouts[layout_i]
14         self.screen.blit(self.backgrounds[layout_i], (0, 0))
```

The `paste()` method pastes a given sprite into a given slot. The slot coordinates come from the current layout.

```
> 15     def paste(self, sprite_i, slot_i):
16         slot = self.slots[slot_i]
17         self.screen.blit(self.sprites[sprite_i], (slot.left, slot.top))
```

The `locate()` method finds the slot index corresponding to a given touch location. It returns the index of the first enclosing slot in the current layout.

```
> 18     def locate(self, x, y):
19         for i, slot in enumerate(self.slots):
20             if slot.left <= x < slot.left + slot.width:
21                 if slot.top <= y < slot.top + slot.height:
22                     return i
```

Recorder.py

This Recorder module is responsible for recording the voter's selections in a tamper-evident, history-independent format.

```
1 import sha
2 class Recorder:
3
```

The **Recorder** object is initialized with access to the ballot definition so it can compute a hash of the ballot data.

```
▷ 4     def __init__(self, ballot):
5         self.hash = sha.new(ballot.data).hexdigest()
```

The **write()** method does the real work of writing out the selections.

```
▷ 6     def write(self, selections):
7         file = open('votes', 'r+')
```

First, the erased portion of the file is skipped. The four-byte sentinel `'\xff\xff\xff\xff'` signals the beginning of the unerased area.

```
8         while file.read(4) != '\xff\xff\xff\xff':
9             pass
```

Then all of the currently stored items are read into the `items` list. Each item is stored as a block of data preceded with the length of the block as a 4-byte unsigned integer. A zero signals that there are no more items.

```
10        items = []
11        size = getint(file)
12        while size:
13            items += [file.read(size)]
14            size = getint(file)
```

Each selection to be written is then encoded as a string of 4-byte integers, preceded by the hash of the ballot definition. These strings are gathered into the `items` list.

```
15        for i, contest in enumerate(selections):
16            for selection in contest:
17                item = self.hash + putint(i)
18                for n in selection:
19                    item += putint(n)
20                items += [item]
```

Sorting the `items` list guarantees a history-independent result.

```
21        items.sort()
```

Next, the size of the region to erase is computed by adding up the maximum possible lengths that each item could have used up, if the items were each added one at a time.

```
22     start = 0
23     maxlength = max([len(item) for item in items] or [''])
24     for i, item in enumerate(items):
25         start += 4 + (4 + maxlength)*i + 4
26
```

The file pointer is then moved to the correct starting location and the new data is written, with the sentinel in front and a zero at the end.

```
27     file.write('\0'*(start - file.tell()))
28     file.seek(start)
29     file.write('\xff\xff\xff\xff')
30     for item in items:
31         file.write(putint(len(item)) + item)
32     file.write(putint(0))
```

After the new data has been successfully written, the region in front of the new data is erased, ensuring an atomic transition from the old data to the new data.

```
33     file.seek(0)
34     file.write('\0'*start)
```

The `getint()` function deserializes an unsigned 4-byte integer from a stream.

```
35 def getint(stream):
36     bytes = [ord(char) for char in stream.read(4)]
37     return (bytes[0]<<24) + (bytes[1]<<16) + (bytes[2]<<8) + bytes[3]
```

The `putint()` function serializes an unsigned integer into a 4-byte string.

```
38 def putint(n):
39     char = lambda n: chr(n & 255)
40     return char(n>>24) + char(n>>16) + char(n>>8) + char(n)
```

B Pvote source code

The following pages present the source code of Pvote, consisting of seven modules:

- `main.py`
- `Ballot.py`
- `verifier.py`
- `Navigator.py`
- `Audio.py`
- `Video.py`
- `Printer.py`

Each line of code is numbered and printed in monospaced type.

```
42      self.bindings = get_list(stream, Binding)
```

Defining occurrences of classes, methods, and functions appear in bold.

```
127  def get_enum(stream, cardinality):
```

Lines marked with a triangle are entry points into a module, called from other modules. Functions and methods without a triangle are called only from within the same module.

```
▷48      def press(self, key):
```

The code is broken into sections, with explanatory text in grey preceding each section.

Explanatory text looks like this.

Reviewers' comments, from the Pvote security review, are marked with bullets and shown in grey italic text after the section to which they refer.

- *Reviewers' notes look like this.*

main.py

This is the main Pvote program. It initializes the other software components with the provided ballot definition file and then processes incoming Pygame events in a non-terminating loop.

```
1 import Ballot, verifier, Audio, Video, Printer, Navigator, pygame
```

These two constants are the type IDs of user-defined events. An AUDIO_DONE event signals that an audio clip has finished playing. A TIMER_DONE event signals that a timed delay has elapsed.

```
2 AUDIO_DONE = pygame.USEREVENT
3 TIMER_DONE = pygame.USEREVENT + 1
```

- *Reviewers suggested that all constants be moved into a separate module; thus, for example, both main.py and Audio.py would refer to the same AUDIO_DONE constant instead of redundantly defining it in both files.*

The following lines load the ballot definition, verify it, and then instantiate the other parts of Pvote with their corresponding sections of the ballot definition.

```
4 ballot = Ballot.Ballot(open("ballot"))
5 verifier.verify(ballot)
6 audio = Audio.Audio(ballot.audio)
7 video = Video.Video(ballot.video)
8 printer = Printer.Printer(ballot.text)
9 navigator = Navigator.Navigator(ballot.model, audio, video, printer)
```

This is the main event loop. The loop begins by updating the display to match the framebuffer in memory, so that any display changes made during the last iteration appear onscreen. The loop never exits.

```
10 while 1:
11     pygame.display.update()
```

On each iteration, one event is retrieved from Pygame's event queue. A timeout is scheduled before waiting for the event, so that if no events occur in `timeout_ms` milliseconds, a `TIMER_DONE` event will be posted. This timeout is then cancelled so that a timer event cannot occur while other processing is taking place.

```
12     pygame.time.set_timer(TIMER_DONE, ballot.model.timeout_ms)
13     event = pygame.event.wait()
14     pygame.time.set_timer(TIMER_DONE, 0)
```

Keypresses are handled by the navigator's `press()` method. Touches on the touchscreen are handled by looking for a corresponding target; if one is found, the event is handled by the navigator's `touch()` method.

```
15     if event.type == pygame.KEYDOWN:
16         navigator.press(event.key)
17     if event.type == pygame.MOUSEBUTTONDOWN:
18         [x, y] = event.pos
19         target_i = video.locate(x, y)
20         if target_i != None:
21             navigator.touch(target_i)
```

The audio driver schedules an `AUDIO_DONE` event to be posted whenever an audio clip finishes playing. Upon receipt of such an event, the audio driver's `next()` method is called so that any audio clips waiting to be played next can start playing.

```
22     if event.type == AUDIO_DONE:
23         audio.next()
```

If a `TIMER_DONE` event was received, that means there has been no user activity for `timeout_ms` milliseconds. It also means that no `AUDIO_DONE` event has occurred for `timeout_ms` milliseconds, which means that either the audio is silent or that a clip has been playing for longer than `timeout_ms` milliseconds. If the `playing` flag on the audio driver is zero, that means the timeout period has elapsed since the last user input occurred or last audio clip finished.

```
24     if event.type == TIMER_DONE and not audio.playing:
25         navigator.timeout()
```

Ballot.py

The `Ballot` module defines the ballot definition data structure. The main program instantiates a `Ballot` object to deserialize the ballot data from a file stream and construct the ballot definition data structure. All the other classes in this module represent parts of the ballot definition; each one deserializes its contents from the stream passed to its constructor.

```
1 import sha
2 class Ballot:
> 3     def __init__(self, stream):
4         assert stream.read(8) == "Pvote\x00\x01\x00"
5         [self.stream, self.sha] = [stream, sha.sha()]
```

In order to produce a SHA-1 hash of all the ballot data, the `Ballot` object passes `self` as the stream object to the other constructors. Its `read` method allows it to proxy for the original stream, allowing it to incorporate all the data into the hash as it passes through. After all four parts of the ballot definition have been loaded, the last 20 bytes of the stream are checked to ensure they match the hash.

```
6         self.model = Model(self)
7         self.text = Text(self)
8         self.audio = Audio(self)
9         self.video = Video(self)
10        assert self.sha.digest() == stream.read(20)
11
12    def read(self, length):
13        data = self.stream.read(length)
14        self.sha.update(data)
15        return data
```

- Reviewers suggested that the `read()` method would make more sense if moved into a separate object playing the role of the stream proxy, instead of using the `Ballot` itself as the stream proxy. This change would also prevent the sub-objects from having access to the incompletely constructed `Ballot` object during construction.

Each remaining class loads its contents from the stream in a constructor that parallels its data structure. These constructors instantiate other classes to read single components from the stream, call `get_list()` to read a variable-length list of components from the stream, or call `get_int()`, `get_enum()`, or `get_str()` to deserialize primitive data types from the stream.

```
15 class Model:
16     def __init__(self, stream):
17         self.groups = get_list(stream, Group)
18         self.pages = get_list(stream, Page)
19         self.timeout_ms = get_int(stream, 0)
20
21 class Group:
22     def __init__(self, stream):
23         self.max_sels = get_int(stream, 0)
24         self.max_chars = get_int(stream, 0)
25         self.option_clips = get_int(stream, 0)
26         self.options = get_list(stream, Option)
```

```

26 class Option:
27     def __init__(self, stream):
28         self.sprite_i = get_int(stream, 0)
29         self.clip_i = get_int(stream, 0)
30         self.writein_group_i = get_int(stream, 1)

31 class Page:
32     def __init__(self, stream):
33         self.bindings = get_list(stream, Binding)
34         self.states = get_list(stream, State)
35         self.option_areas = get_list(stream, OptionArea)
36         self.counter_areas = get_list(stream, CounterArea)
37         self.review_areas = get_list(stream, ReviewArea)

38 class State:
39     def __init__(self, stream):
40         self.sprite_i = get_int(stream, 0)
41         self.segments = get_list(stream, Segment)
42         self.bindings = get_list(stream, Binding)
43         self.timeout_segments = get_list(stream, Segment)
44         self.timeout_page_i = get_int(stream, 1)
45         self.timeout_state_i = get_int(stream, 0)

46 class OptionArea:
47     def __init__(self, stream):
48         self.group_i = get_int(stream, 0)
49         self.option_i = get_int(stream, 0)

50 class CounterArea:
51     def __init__(self, stream):
52         self.group_i = get_int(stream, 0)
53         self.sprite_i = get_int(stream, 0)

54 class ReviewArea:
55     def __init__(self, stream):
56         self.group_i = get_int(stream, 0)
57         self.cursor_sprite_i = get_int(stream, 1)

58 class Binding:
59     def __init__(self, stream):
60         self.key = get_int(stream, 1)
61         self.target_i = get_int(stream, 1)
62         self.conditions = get_list(stream, Condition)
63         self.steps = get_list(stream, Step)
64         self.segments = get_list(stream, Segment)
65         self.next_page_i = get_int(stream, 1)
66         self.next_state_i = get_int(stream, 0)

67 class Condition:
68     def __init__(self, stream):
69         self.predicate = get_enum(stream, 3)
70         self.group_i = get_int(stream, 1)
71         self.option_i = get_int(stream, 0)
72         self.invert = get_enum(stream, 2)

73 class Step:
74     def __init__(self, stream):
75         self.op = get_enum(stream, 5)
76         self.group_i = get_int(stream, 1)
77         self.option_i = get_int(stream, 0)

```

```

78 class Segment:
79     def __init__(self, stream):
80         self.conditions = get_list(stream, Condition)
81         self.type = get_enum(stream, 5)
82         self.clip_i = get_int(stream, 0)
83         self.group_i = get_int(stream, 1)
84         self.option_i = get_int(stream, 0)

85 class Text:
86     def __init__(self, stream):
87         self.groups = get_list(stream, TextGroup)

88 class TextGroup:
89     def __init__(self, stream):
90         self.name = get_str(stream)
91         self.writein = get_enum(stream, 2)
92         self.options = get_list(stream, get_str)

93 class Audio:
94     def __init__(self, stream):
95         self.sample_rate = get_int(stream, 0)
96         self.clips = get_list(stream, Clip)

```

The **Clip** type contains the waveform data for an audio clip, which resides in a single Python string. In a serialized ballot definition, the number of samples is stored preceding the audio data. Since each sample is a 16-bit value, the number of bytes to read is twice the number of samples.

```

97 class Clip:
98     def __init__(self, stream):
99         self.samples = stream.read(get_int(stream, 0)*2)

100 class Video:
101     def __init__(self, stream):
102         self.width = get_int(stream, 0)
103         self.height = get_int(stream, 0)
104         self.layouts = get_list(stream, Layout)
105         self.sprites = get_list(stream, Image)

106 class Layout:
107     def __init__(self, stream):
108         self.screen = Image(stream)
109         self.targets = get_list(stream, Rect)
110         self.slots = get_list(stream, Rect)

```

An **Image** object contains the pixel data for an image, which resides in a single Python string. In serialized form, the image's width and height are stored preceding the pixel data, which contains three bytes per pixel (one byte each for the red, green, and blue components).

```

111 class Image:
112     def __init__(self, stream):
113         self.width = get_int(stream, 0)
114         self.height = get_int(stream, 0)
115         self.pixels = stream.read(self.width*self.height*3)

116 class Rect:
117     def __init__(self, stream):
118         self.left = get_int(stream, 0)
119         self.top = get_int(stream, 0)
120         self.width = get_int(stream, 0)
121         self.height = get_int(stream, 0)

```

The `get_int()` function reads an unsigned 4-byte integer from the stream. The `allow_none` argument is a flag specifying whether the returned value can be `None`, which is represented by the sequence `"\xff\xff\xff\xff"`. This function ensures that the data meets the constraints given in the assurance document—namely, that the value is between 0 and $2^{31} - 1$ inclusive, or `None` only for fields that allow it.

```
122 def get_int(stream, allow_none):
123     [a, b, c, d] = list(stream.read(4))
124     if ord(a) < 128:
125         return ord(a)*16777216 + ord(b)*65536 + ord(c)*256 + ord(d)
126     assert allow_none and a + b + c + d == "\xff\xff\xff\xff"
```

- Reviewers suggested that it would be clearer to have two separate methods (for reading an integer and reading an integer-or-None) instead of using `get_int()` for both purposes.
- Reviewers agreed that there should be an explicit `return None` statement to show that `None` is the intended return value.

The `get_enum()` function reads an enumerated type from the stream, which is represented the same way as an integer. The second argument gives the cardinality of the enumeration, which is used to ensure the validity of the returned value.

```
127 def get_enum(stream, cardinality):
128     value = get_int(stream, 0)
129     assert value < cardinality
130     return value
```

- Reviewers suggested that it would be clearer to have two separate methods for reading Boolean values and enumerated values, instead of using `get_enum(stream, 2)` to read Boolean values.

The `get_str()` function reads a string from the stream, which is represented as a sequence of bytes prefixed by the length as a 4-byte integer. This function checks that all the characters in the string fall in the printable ASCII range, so they will print out in a predictable way. The tilde character (number 126) is specifically excluded to avoid any ambiguity in the printed output, because the tilde is used as a delimiter.

```
131 def get_str(stream):
132     str = stream.read(get_int(stream, 0))
133     for ch in list(str):
134         assert 32 <= ord(ch) <= 125
135     return str
```

- Reviewers suggested that the condition in line 134 would be easier to understand if it were written `isprint(ch)` and `ch != '~'`.

The `get_list()` function reads a variable-length list of data structures from the stream, all of a particular given class. In Python (and Pthin), classes are first-class objects and can be passed as arguments. In serialized form, the list is preceded by a 4-byte integer indicating how many elements to read.

```
136 def get_list(stream, Class):
137     return [Class(stream) for i in range(get_int(stream, 0))]
```

verifier.py

The `verifier` module contains only one entry point, `verify()`, whose responsibility is to abort the program if the ballot definition is not well-formed. The intention is that, if execution continues after a call to `verify()`, it should never abort thereafter—that is: (a) `verify()` checks all the assumptions about the ballot definition upon which the rest of `Pvote` relies; and (b) the contents of the ballot definition data structures are never changed after `verify()` is called.

```
▷ 1 def verify(ballot):
  2     [groups, sprites] = [ballot.model.groups, ballot.video.sprites]
```

`option_sizes` contains one list corresponding to each group; it will collect all the sprites for the options in that group and all the slots in which such options could be pasted (in option areas and review areas). `char_sizes` also contains one list for each group; it will collect all the sprites for characters corresponding to write-in options in the group, as well as all the slots in which such characters could be pasted (in review areas). These lists will later be checked to ensure that the sizes of all sprites match the sizes of all the slots into which they could be pasted.

```
  3     option_sizes = [[] for group in groups]
  4     char_sizes = [[] for group in groups]
```

The following lines ensure that the parallel arrays have matching size. It also makes sure that they are also nonempty; for example, the navigator assumes that there is at least one page when it starts up with a transition to page 0.

```
  5     assert len(ballot.model.groups) == len(ballot.text.groups) > 0
  6     assert len(ballot.model.pages) == len(ballot.video.layouts) > 0
```

For each page, the list of bindings are checked. Each page also has to have at least one state.

```
  7     for [page_i, page] in enumerate(ballot.model.pages):
  8         layout = ballot.video.layouts[page_i]
  9
 10         for binding in page.bindings:
 11             verify_binding(ballot, page, binding)
 12         assert len(page.states) > 0
```

For each state, the segments and bindings are checked. The sprite is checked to make sure it exactly fills its slot, and the timeout transition is also checked for validity.

```
 12         for [state_i, state] in enumerate(page.states):
 13             verify_size(sprites[state.sprite_i], layout.slots[state_i])
 14             verify_segments(ballot, page, state.segments)
 15             for binding in state.bindings:
 16                 verify_binding(ballot, page, binding)
 17             verify_segments(ballot, page, state.timeout_segments)
 18             verify_goto(ballot, state.timeout_page_i, state.timeout_state_i)
 19             slot_i = len(page.states)
```

Each option area is checked for a valid option reference, and the option slots are gathered into the appropriate array for later size checking.

```
20     for area in page.option_areas:
21         verify_option_ref(ballot, page, area)
22         option_sizes[area.group_i].append(layout.slots[slot_i])
23         slot_i = slot_i + 1
```

For each counter area, all the possible sprites that could be pasted are checked to ensure they exactly fill the slot.

```
24     for area in page.counter_areas:
25         for i in range(groups[area.group_i].max_sels + 1):
26             verify_size(sprites[area.sprite_i + i], layout.slots[slot_i])
27             slot_i = slot_i + 1
```

For each review area, the slots for options and characters are gathered into the appropriate array for later size checking. If there is a cursor sprite, its size is expected to match the option slots as well.

```
28     for area in page.review_areas:
29         for i in range(groups[area.group_i].max_sels):
30             option_sizes[area.group_i].append(layout.slots[slot_i])
31             slot_i = slot_i + 1
32         for j in range(groups[area.group_i].max_chars):
33             char_sizes[area.group_i].append(layout.slots[slot_i])
34             slot_i = slot_i + 1
35         if area.cursor_sprite_i != None:
36             option_sizes[area.group_i].append(sprites[area.cursor_sprite_i])
```

The sprites for all the options and characters are gathered into the appropriate arrays. The audio clip indices for the options are ensured to be within range. For write-in options, the number of allowed write-in characters in the parent group is checked to ensure it matches the number of allowed selections in the write-in group; thus, all the write-in options in a group are required to accept the same number of characters. Write-in groups are not themselves allowed to contain write-ins.

```
37     for [group_i, group] in enumerate(groups):
38         for option in group.options:
39             option_sizes[group_i].append(sprites[option.sprite_i])
40             option_sizes[group_i].append(sprites[option.sprite_i + 1])
41         assert group.option_clips > 0
42         ballot.audio.clips[option.clip_i + group.option_clips - 1]
43         if option.writein_group_i != None:
44             writein_group = groups[option.writein_group_i]
45             assert writein_group.max_chars == 0
46             assert writein_group.max_sels == group.max_chars > 0
47             for option in writein_group.options:
48                 char_sizes[group_i].append(sprites[option.sprite_i])
```

The sprites and slots that have been collected for each group are now checked to ensure they all have matching sizes.

```
49     for object in option_sizes[group_i]:
50         verify_size(object, option_sizes[group_i][0])
51     for object in char_sizes[group_i]:
52         verify_size(object, char_sizes[group_i][0])
```

The text section is checked to ensure that every option has a name, and ensure that the group names and option names have reasonable lengths that will print properly.

```
53     for [group_i, group] in enumerate(ballot.text.groups):
54         assert len(group.name) <= 50
55         assert len(group.options) == len(groups[group_i].options)
56         for option in group.options:
57             assert len(option) <= 50
```

Every audio clip is checked to ensure that it has nonzero length. There is no Pvote code that relies on this property; Pygame has the an unfortunate limitation that the audio system will abort if asked to play a zero-length sound.

```
58     for clip in ballot.audio.clips:
59         assert len(clip.samples) > 0
```

Finally, the video section is checked. The background images must match the screen size, all the slots and targets must fit entirely onscreen, and the image data for each sprite must match the sprite's claimed dimensions.

```
60     assert ballot.video.width*ballot.video.height > 0
61     for layout in ballot.video.layouts:
62         verify_size(layout.screen, ballot.video)
63         for rect in layout.targets + layout.slots:
64             assert rect.left + rect.width <= ballot.video.width
65             assert rect.top + rect.height <= ballot.video.height
66     for sprite in ballot.video.sprites:
67         assert len(sprite.pixels) == sprite.width*sprite.height*3 > 0
```

The `verify_binding()` function checks that a binding is well-formed by inspecting each of its parts: its list of conditions, its list of steps, its list of audio segments, and its transition.

```
68 def verify_binding(ballot, page, binding):
69     for condition in binding.conditions:
70         verify_option_ref(ballot, page, condition)
71     for step in binding.steps:
72         verify_option_ref(ballot, page, step)
73     verify_segments(ballot, page, binding.segments)
74     verify_goto(ballot, binding.next_page_i, binding.next_state_i)
```

The `verify_goto()` function checks that the page index and state index for a transition are within range. None is an allowed value for the page index.

```
75 def verify_goto(ballot, page_i, state_i):
76     if page_i != None:
77         ballot.model.pages[page_i].states[state_i]
```

The `verify_segments()` function checks that a list of segments is well-formed. It inspects each segment's list of conditions and, based on the segment type, ensures that all the possible corresponding indices of audio clips are within range.

```
78 def verify_segments(ballot, page, segments):
79     for segment in segments:
80         for condition in segment.conditions:
81             verify_option_ref(ballot, page, condition)
82             ballot.audio.clips[segment.clip_i]
83             if segment.type in [1, 2, 3, 4]:
84                 group = verify_option_ref(ballot, page, segment)
85                 if segment.type in [1, 2]:
86                     assert segment.clip_i < group.option_clips
87                 if segment.type in [3, 4]:
88                     ballot.audio.clips[segment.clip_i + group.max_sels]
```

Reviewers wanted to see meaningfully named constants here for the enumerated values. They recommended that all the enumerated value constants should be pulled out into a separate module—thus, for example, the above code and the navigator code would refer to the same set of `SG_*` constants.

The `verify_option_ref()` function checks the validity of an (indirect or direct) option reference in a condition, step, or segment—all of these types have a `group_i` field and an `option_i` field. If the `group_i` field is `None`, then `option_i` must be the index of a valid option area on the current page. Otherwise, `group_i` and `option_i` must be valid group and option indices respectively. The group object is returned as a convenience for `verify_segments()`, which uses the group object for other checks.

```
89 def verify_option_ref(ballot, page, object):
90     if object.group_i == None:
91         area = page.option_areas[object.option_i]
92         return ballot.model.groups[area.group_i]
93     return ballot.model.groups[object.group_i].options[object.option_i]
```

The `verify_size()` function ensures that two objects (sprites or slots) have the same dimensions.

```
95 def verify_size(a, b):
96     assert a.width == b.width and a.height == b.height
```

Navigator.py

The first three lines set up constants corresponding to the three enumerated types in the ballot model definition: `OP_*` for step types, `SG_*` for audio segment types, and `PR_*` for predicates in conditions.

```
1 [OP_ADD, OP_REMOVE, OP_APPEND, OP_POP, OP_CLEAR] = range(5)
2 [SG_CLIP, SG_OPTION, SG_LIST_SELS, SG_COUNT_SELS, SG_MAX_SELS] = range(5)
3 [PR_GROUP_EMPTY, PR_GROUP_FULL, PR_OPTION_SELECTED] = range(3)
```

The navigator is initialized with access to the ballot model data structure, audio driver, video driver, and printing module. It saves these references locally, initializes an empty selection state, and begins the voting session by transitioning to state 0 of page 0.

```
4 class Navigator:
> 5     def __init__(self, model, audio, video, printer):
6         self.model = model
7         [self.audio, self.video, self.printer] = [audio, video, printer]
8         self.selections = [[] for group in model.groups]
9         self.page_i = None
10        self.goto(0, 0)
```

The `goto()` method transitions to a given state and page. It is called by `invoke()` and `timeout()`. If the transition goes to the last page, the voter's selections are committed. Any state transition (even a transition back to the current state) triggers the playback of the state's audio segments; the `play()` method queues the audio instantaneously for later playback. In the ballot definition, `page_i` can be `None` to indicate that no transition should occur; that case is accepted and handled here. Other methods rely on `goto()` to always update the video display with a call to `update()`, even if no state transition occurs.

```
11     def goto(self, page_i, state_i):
12         if page_i != None and self.page_i != len(self.model.pages) - 1:
13             if page_i == len(self.model.pages) - 1:
14                 self.printer.write(self.selections)
15                 [self.page_i, self.page] = [page_i, self.model.pages[page_i]]
16                 [self.state_i, self.state] = [state_i, self.page.states[state_i]]
17                 self.play(self.state.segments)
18                 self.update()
```

• Reviewers found the logic of line 12 confusing, as it combines the “no transition” condition with the “already committed” condition. They all agreed that the navigator should have a flag that indicates whether the votes have already been committed, and a separate method that commits the votes and sets the flag. They also suggested that, to make the commit condition more obvious, the navigator should start on page 1 and always commit on page 0.

The `update()` method updates the video display based on the current page, state, and selections. It tells the video driver to paste the page's background image over the entire screen, then lay the state's sprite on top of that, and finally fills in any option areas, counter areas, and review areas on the page, in that order. The indices of the slots are assumed to be arranged in sequential order, as described in Chapter 7; hence the variable `slot_i` is incremented in each loop and carried forward to the next loop. Because review areas occupy a variable number of slots depending on their group, the review area loop relies on the `review()` method to return an appropriately incremented value for `slot_i`.

```

19     def update(self):
20         self.video.goto(self.page_i)
21         self.video.paste(self.state.sprite_i, self.state_i)

22         slot_i = len(self.page.states)
23         for area in self.page.option_areas:
24             unselected = area.option_i not in self.selections[area.group_i]
25             group = self.model.groups[area.group_i]
26             option = group.options[area.option_i]
27             self.video.paste(option.sprite_i + unselected, slot_i)
28             slot_i = slot_i + 1

29         for area in self.page.counter_areas:
30             count = len(self.selections[area.group_i])
31             self.video.paste(area.sprite_i + count, slot_i)
32             slot_i = slot_i + 1

33         for area in self.page.review_areas:
34             slot_i = self.review(area.group_i, slot_i, area.cursor_sprite_i)

```

The `review()` method fills in the appropriate sprites for a review area. The arguments `group_i` and `cursor_sprite_i` are parameters of the review area; `slot_i` should be the index of the review area's first slot. The main loop always runs `group.max_sels` times to ensure that `slot_i` cannot go out of range, and that `slot_i` is incremented by the correct amount: `max_sels × (1 + max_chars)`. Each selected option is pasted into a slot, and then, if the option is a write-in option, a recursive call to `review()` fills in the characters of the write-in. If a cursor sprite is given, it is pasted into the slot just after the last selected option.

```

35     def review(self, group_i, slot_i, cursor_sprite_i):
36         group = self.model.groups[group_i]
37         selections = self.selections[group_i]
38         for i in range(group.max_sels):
39             if i < len(selections):
40                 option = group.options[selections[i]]
41                 self.video.paste(option.sprite_i, slot_i)
42                 if option.writein_group_i != None:
43                     self.review(option.writein_group_i, slot_i + 1, None)
44             if i == len(selections) and cursor_sprite_i != None:
45                 self.video.paste(cursor_sprite_i, slot_i)
46             slot_i = slot_i + 1 + group.max_chars
47         return slot_i

```

- *The reviewers generally found this method to be the most confusing part of the source code, because of its use of recursion and the arithmetic involved in determining `slot_i`. They suggested splitting this into two methods such as `review_contest()` and `review_writein()`; `review_contest()` would call `review_writein()` when necessary. Even though there would be substantial duplication between the two methods, the reviewers felt that eliminating recursion was more important.*

The `press()` and `touch()` methods handle incoming events from the main loop: `press()` handles keypresses and `touch()` handles screen touches. Both methods scan through the bindings of the current state and page, searching for a binding that matches the pressed key or touched target and whose conditions are all satisfied. The first such binding (and only the first such binding) is invoked with a call to the `invoke()` method.

```
▷ 48     def press(self, key):
49         for binding in self.state.bindings + self.page.bindings:
50             if key == binding.key and self.test(binding.conditions):
51                 return self.invoke(binding)

▷ 52     def touch(self, target_i):
53         for binding in self.state.bindings + self.page.bindings:
54             if target_i == binding.target_i and self.test(binding.conditions):
55                 return self.invoke(binding)
```

- Reviewers felt the method names `press()` and `touch()` were too similar and could be made clearer.

The `test()` method evaluates a list of conditions and returns 1 only if all the conditions are met. Each of the three predicate types is evaluated in a separate clause; the `cond.invert` flag indicates whether to invert the sense of an individual predicate.

```
56     def test(self, conditions):
57         for cond in conditions:
58             [group_i, option_i] = self.get_option(cond)
59             if cond.predicate == PR_GROUP_EMPTY:
60                 result = len(self.selections[group_i]) == 0
61             if cond.predicate == PR_GROUP_FULL:
62                 max = self.model.groups[group_i].max_sels
63                 result = len(self.selections[group_i]) == max
64             if cond.predicate == PR_OPTION_SELECTED:
65                 result = option_i in self.selections[group_i]
66             if cond.invert == result:
67                 return 0
68         return 1
```

- Reviewers felt the comparison of Boolean values on line 66 was “just too clever for its own good.” They agreed that lines 66 and 67 could have been more clearly written as

```
    if cond.invert:
        result = not result
    if not result:
        return 0
```

to show that `cond.invert` reverses the sense of the condition and that the loop body returns 0 only when the condition is not met.

The `invoke()` method invokes a binding. The steps of the action are carried out, then the audio for the binding is queued, and finally the state transition, if any, takes place. (The `goto()` method handles the case where `next_page_i` is None.) Invoking a binding always interrupts any currently playing audio.

```
69     def invoke(self, binding):
70         for step in binding.steps:
71             self.execute(step)
72         self.audio.stop()
73         self.play(binding.segments)
74         self.goto(binding.next_page_i, binding.next_state_i)
```

The `execute()` method executes a single step, which operates on the selection state. It is responsible for ensuring that invalid selection states are never reached.

```
75     def execute(self, step):
76         [group_i, option_i] = self.get_option(step)
77         group = self.model.groups[group_i]
78         selections = self.selections[group_i]
79         selected = option_i in selections
80
81         if step.op == OP_ADD and not selected or step.op == OP_APPEND:
82             if len(selections) < group.max_sels:
83                 selections.append(option_i)
84             if step.op == OP_REMOVE and selected:
85                 selections.remove(option_i)
86
87         if step.op == OP_POP and len(selections) > 0:
88             selections.pop()
89         if step.op == OP_CLEAR:
90             self.selections[group_i] = []
```

- Reviewers felt the Boolean expression on line 80 should be clarified with parentheses.
- Reviewers found the `execute()` method more confusing than necessary because it uses both the list `self.selections` and a local variable `selections` that aliases a part of it. Mixing these two ways of accessing the list makes it harder to reason about the code, because each could have side-effects on the other. The method would be easier to verify if it always accessed the list through just `self.selections` or just `selections`.
- Reviewers felt the method names `invoke()` and `execute()` were too similar and could be made clearer.

The `timeout()` method handles an inactivity timeout. It is called by the main event loop.

```
> 89     def timeout(self):
90         self.play(self.state.timeout_segments)
91         self.goto(self.state.timeout_page_i, self.state.timeout_state_i)
```

The `play()` method plays a list of audio segments. Its job is to translate a list of segments into a sequence of audio clip indices, and send these indices to the audio driver to be queued for playing. Each segment's conditions are checked; if the conditions are met, the corresponding clip index (or indices) are sent to the audio driver. After the clips are queued, `play()` returns immediately; it does not wait for the audio to finish playing, or even to start playing.

```

92     def play(self, segments):
93         for segment in segments:
94             if self.test(segment.conditions):
95                 if segment.type == SG_CLIP:
96                     self.audio.play(segment.clip_i)
97             else:
98                 [group_i, option_i] = self.get_option(segment)
99                 group = self.model.groups[group_i]
100                selections = self.selections[group_i]

101                if segment.type == SG_OPTION:
102                    self.play_option(group.options[option_i], segment.clip_i)
103                if segment.type == SG_LIST_SELS:
104                    for option_i in selections:
105                        self.play_option(group.options[option_i], segment.clip_i)
106                if segment.type == SG_COUNT_SELS:
107                    self.audio.play(segment.clip_i + len(selections))
108                if segment.type == SG_MAX_SELS:
109                    self.audio.play(segment.clip_i + group.max_sels)

```

The `play_option()` method sends audio clips for a given option to the audio driver. There can be multiple clips associated with each option, as dictated by the `option_clips` field of its containing group; the `offset` argument selects which one to play. For a write-in option, this entails playing, in sequence, all the audio clips for the characters in the write-in. Write-in characters are assumed to have only one clip each.

```

110     def play_option(self, option, offset):
111         self.audio.play(option.clip_i + offset)
112         if option.writein_group_i != None:
113             writein_group = self.model.groups[option.writein_group_i]
114             for option_i in self.selections[option.writein_group_i]:
115                 self.audio.play(writein_group.options[option_i].clip_i)

```

The `get_option()` method is used by `test()`, `execute()`, and `play()` to determine the specific group and option for a condition, step, or segment respectively. Conditions, steps, and segments all have fields named `group_i` and `option_i` that can refer to an option either directly or indirectly. When `group_i` is `None`, it's an indirect reference: `option_i` is the index of an option area on the current page. When `group_i` is not `None`, it's a direct reference: `group_i` and `option_i` specify the intended option.

```

116     def get_option(self, object):
117         if object.group_i == None:
118             area = self.page.option_areas[object.option_i]
119             return [area.group_i, area.option_i]
120         return [object.group_i, object.option_i]

```

Audio.py

Audio playback is provided by the pygame library.

```
1 import pygame
```

Pygame is based on an event-loop control model. Instead of invoking callbacks, Pygame queues events for processing by the application. Each event has an integer type ID, and Pygame supports user-defined events with type IDs equal to `pygame.USEREVENT` or higher. This module uses `AUDIO_DONE` for signalling when an audio clip has finished playing.

```
2 AUDIO_DONE = pygame.USEREVENT
```

- *Reviewers suggested that constants like these all be collected in a separate module, and that `main.py` and `Audio.py` refer to the same `AUDIO_DONE` constant instead of redundantly defining it in both files.*

The `Audio` class is responsible for maintaining a queue of audio clips and causing them to be played in sequence. It ensures that only one clip is playing at a time, and that all the clips are played back one after another until the queue is empty.

```
3 class Audio:
```

The audio driver is initialized with access to the audio section of the ballot definition. It initializes the Pygame audio mixer and converts all the audio clips from raw data into Pygame `Sound` objects. The `playing` flag is exposed to the main program; it indicates whether or not audio is currently playing.

```
> 4     def __init__(self, audio):
5         rate = audio.sample_rate
6         pygame.mixer.init(rate, -16, 0)
7         self.clips = [make_sound(rate, clip.samples) for clip in audio.clips]
8         [self.queue, self.playing] = [[], 0]
```

The `play()` method puts a single audio clip on the queue. If nothing is currently playing, playback of the given audio clip immediately begins.

```
> 9     def play(self, clip_i):
10         self.queue.append(clip_i)
11         if not self.playing:
12             self.next()
```

The `next()` method takes the next available audio clip off of the queue and starts playing it. The `AUDIO_DONE` event is scheduled to be posted when the audio clip finishes playing. The `playing` member is set to a nonzero value if and only if an audio clip is playing.

```
> 13     def next(self):
14         self.playing = len(self.queue)
15         if len(self.queue):
16             self.clips[self.queue.pop(0)].play().set_endevent(AUDIO_DONE)
```

The `stop()` method stops audio playback and cancels pending audio.

```
> 17     def stop(self):
18         self.queue = []
19         pygame.mixer.stop()
```

The `make_sound()` function converts a string of audio data into a Pygame **Sound** object. Because Pygame only knows how to load sounds from files, and the only uncompressed sound format that Pygame accepts is the Microsoft WAVE format, we have to construct a fake file object with a WAVE file header. The header always specifies no compression, monaural audio, and signed 16-bit samples.

```
20 def make_sound(rate, data):
21     [comp_channels, sample_size] = ["\x01\x00\x01\x00", "\x02\x00\x10\x00"]
22     fmt = comp_channels + put_int(rate) + put_int(rate*2) + sample_size
23     file = chunk("RIFF", "WAVE" + chunk("fmt ", fmt) + chunk("data", data))
24     return pygame.mixer.Sound(Buffer(file))
```

The `chunk()` function creates a RIFF chunk, which consists of a 4-byte type code and a 4-byte length followed by a string of data.

```
25 def chunk(type, contents):
26     return type + put_int(len(contents)) + contents
```

The `put_int()` function converts an integer into a 4-byte big-endian representation.

```
27 def put_int(n):
28     [a, b, c, d] = [n/16777216, n/65536, n/256, n]
29     return chr(d % 256) + chr(c % 256) + chr(b % 256) + chr(a % 256)
```

The `Buffer` class is a thin wrapper that makes a string look like a readable file. `make_sound()` wraps this class around the WAVE formatted audio data so it can be passed to Pygame to create a **Sound** object.

```
30 class Buffer:
31     def __init__(self, data):
32         [self.data, self.pos] = [data, 0]
33
34     def read(self, length):
35         self.pos = self.pos + length
36         return self.data[self.pos - length:self.pos]
```

Video.py

Video display control is provided by the pygame library.

```
1 import pygame
```

The `make_image()` function converts a string containing uncompressed pixel data into a Pygame `Image` object.

```
2 def make_image(im):
3     return pygame.image.fromstring(im.pixels, (im.width, im.height), "RGB")
```

The `Video` class is responsible for pasting full-screen images and sprites onto the display, as well as translating touch locations into target indices.

```
4 class Video:
```

The video driver is initialized with access to the video section of the ballot definition. It initializes the Pygame display and converts all the images from raw data into Pygame `Image` objects. The video driver keeps a pointer to the current layout in its `layout` member so it can look up slots and targets for the current page.

```
> 5     def __init__(self, video):
6         size = [video.width, video.height]
7         self.surface = pygame.display.set_mode(size, pygame.FULLSCREEN)
8         self.layouts = video.layouts
9         self.screens = [make_image(layout.screen) for layout in video.layouts]
10        self.sprites = [make_image(sprite) for sprite in video.sprites]
11        self.goto(0)
```

The `goto()` method switches to a given layout, which involves pasting the layout's background image over the entire screen.

```
> 12     def goto(self, layout_i):
13         self.layout = self.layouts[layout_i]
14         self.surface.blit(self.screens[layout_i], [0, 0])
```

The `paste()` method pastes a given sprite into a given slot. The slot coordinates are looked up in the current layout.

```
> 15     def paste(self, sprite_i, slot_i):
16         slot = self.layout.slots[slot_i]
17         self.surface.blit(self.sprites[sprite_i], [slot.left, slot.top])
```

The `locate()` method finds the target index corresponding to a given touch location. It returns the index of the first enclosing target in the current layout.

```
> 18     def locate(self, x, y):
19         for [i, target] in enumerate(self.layout.targets):
20             if target.left <= x and x < target.left + target.width:
21                 if target.top <= y and y < target.top + target.height:
22                     return i
```

Printer.py

The **Printer** class commits the voter's selections by printing them out. (Other vote-recording mechanisms could be substituted for this module.) It is initialized with access to the text section of the ballot definition.

```
1 class Printer:
2     def __init__(self, text):
3         self.text = text
```

The **write()** method does the printing, assuming that the standard output stream is connected to a printer. To prevent any possibility of ambiguous output, the first character of every printed line indicates its purpose, and lines never wrap. An asterisk (*) marks a contest, and a minus sign (-) marks an option. A plus sign (+) marks a write-in group, and an equals sign (=) marks the text of the write-in. A tilde (~) is printed after the name of each write-in character because characters can have names of any length (a feature intended to let ASCII printouts describe write-ins containing non-ASCII characters.) A tilde on a line by itself marks the end of the printout. Here is an example of a printout:

```
* Governor
- Peter Miguel Camejo

* Secretary of State ~ NO SELECTION

* Member of City Council
- William "Bill" G. Glynn
- Write-in 1

+ Member of City Council, Write-in 1
= S~T~E~P~H~E~N~ ~H~A~W~K~I~N~G~

* Proposition 1A
- Yes

~

▷ 4     def write(self, selections):
5         for [group_i, selection] in enumerate(selections):
6             group = self.text.groups[group_i]
7             if group.writein:
8                 if len(selection):
9                     print "\n+ " + group.name
10                    line = ""
11                    for option_i in selection:
12                        if len(line) + len(group.options[option_i]) + 1 > 60:
13                            print "= " + line
14                            line = ""
15                            line = line + group.options[option_i] + "~"
16                    print "= " + line
17             else:
18                 if len(selection):
19                     print "\n* " + group.name
20                     for [option_i, option] in enumerate(group.options):
21                         if option_i in selection:
22                             print "- " + option
23                 else:
24                     print "\n* " + group.name + " NO SELECTION"
25         print "\n~\f"
```

C Sample Pvote ballot definition

This appendix describes the construction of a ballot definition file for Pvote (the same ballot file mentioned on page 133). It is based on ballot style #167 for the November 2006 election in Contra Costa County, California. The paper ballot has 16 elected offices, 12 judicial confirmations, and 16 referenda. This ballot definition just contains the first two state offices (Governor and Secretary of State), one local office (City Council), and two state measures (Propositions 1A and 1B).

This sample ballot definition is not intended to serve as an example of optimally usable or optimally accessible ballot design. It is merely intended to demonstrate a few different interaction models that are achievable with Pvote, and to make a plausible case that it is possible to design a single ballot definition file that works for voters who use only the visual interface, voters who use only the audio interface, or voters who use the visual and audio interfaces together.

Audio messages are shown in a sans-serif typeface. Boxes indicate variable parts of the message. When a series of boxes are joined by dashes, one box in the series is played depending on the voter's current selections. A box can also contain text in italics describing the message to be played. Here is an example:

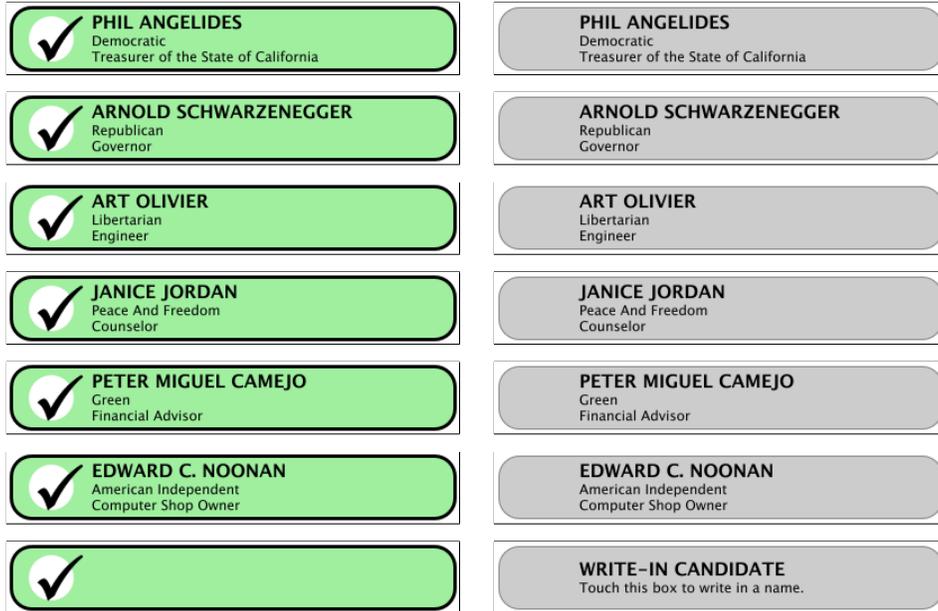
Please vote for one. No choices are currently selected.—
Your current selection is *list of selected options*.

The above describes an audio message consisting of:

- First, the spoken message “Please vote for one.”
- Then, either the spoken message “No choices are currently selected.” or the message “Your current selection is.”
- Finally, a spoken list of the selected options.

There are 10 groups and 17 pages in this ballot definition. The groups are as follows.

Group 0. This is the contest for Governor, with `max_sels = 1`, `max_chars = 25`, and `option_clips = 2`. It contains 7 options. There are two sprites for each option:

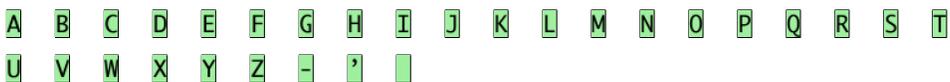


Each option has two associated audio clips, for a short and a long spoken description. For example, option 0 has the two clips:

- Phil Angelides
- Phil Angelides. Democratic Party. Treasurer of the State of California.

The last option, option 6, has `writein_group = 1`; the rest have `writein_group = None`.

Group 1. This is the write-in group for the Governor contest, with `max_sels = 25`, `max_chars = 0`, and `option_clips = 1`. It has 29 options, with the sprites:



Each option has one associated audio clip with the name of the character (the names of the letters of the alphabet and the spoken words “hyphen”, “apostrophe”, and “space”).

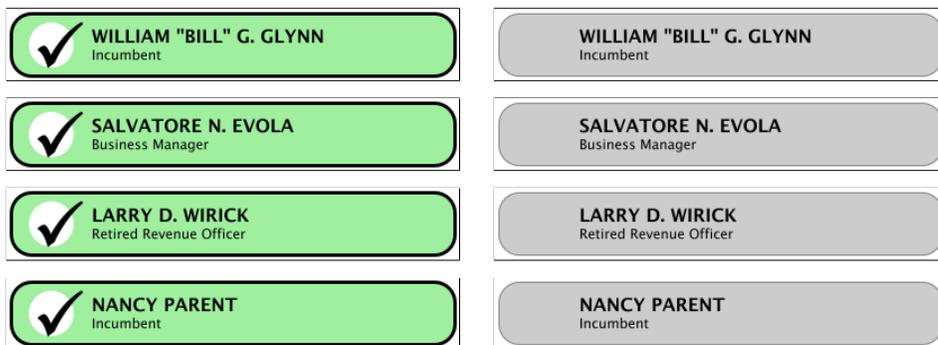
Group 2. This is the contest for Secretary of State, with `max_sels = 1`, `max_chars = 25`, and `option_clips = 2`. It contains 7 options, with two sprites for each option:



Just as in group 0, each option has two associated audio clips giving a short and a long spoken description. The last option, option 6, has `writein_group = 3`; the rest have `writein_group = None`.

Group 3. This is the write-in group for the Secretary of State contest, with `max_sels = 25`, `max_chars = 0`, and `option_clips = 1`. It has the same options as group 1.

Group 4. This is the contest for City Council, with `max_sels = 3`, `max_chars = 25`, and `option_clips = 2`. It contains 8 options, with two sprites for each option:

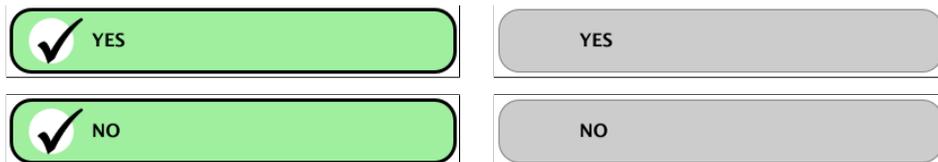




Just as in groups 0 and 2, each option has two associated audio clips giving a short and a long spoken description. Each of the last three options has its own write-in group: option 5 has `writeln_group = 5`, option 6 has `writeln_group = 6`, and option 7 has `writeln_group = 7`. The rest of the options have `writeln_group = None`.

Groups 5, 6, and 7. These are the write-in groups for the three write-in options in the City Council contest. All of them have `max_sels = 25`, `max_chars = 0`, `option_clips = 1`, and the same options as group 1.

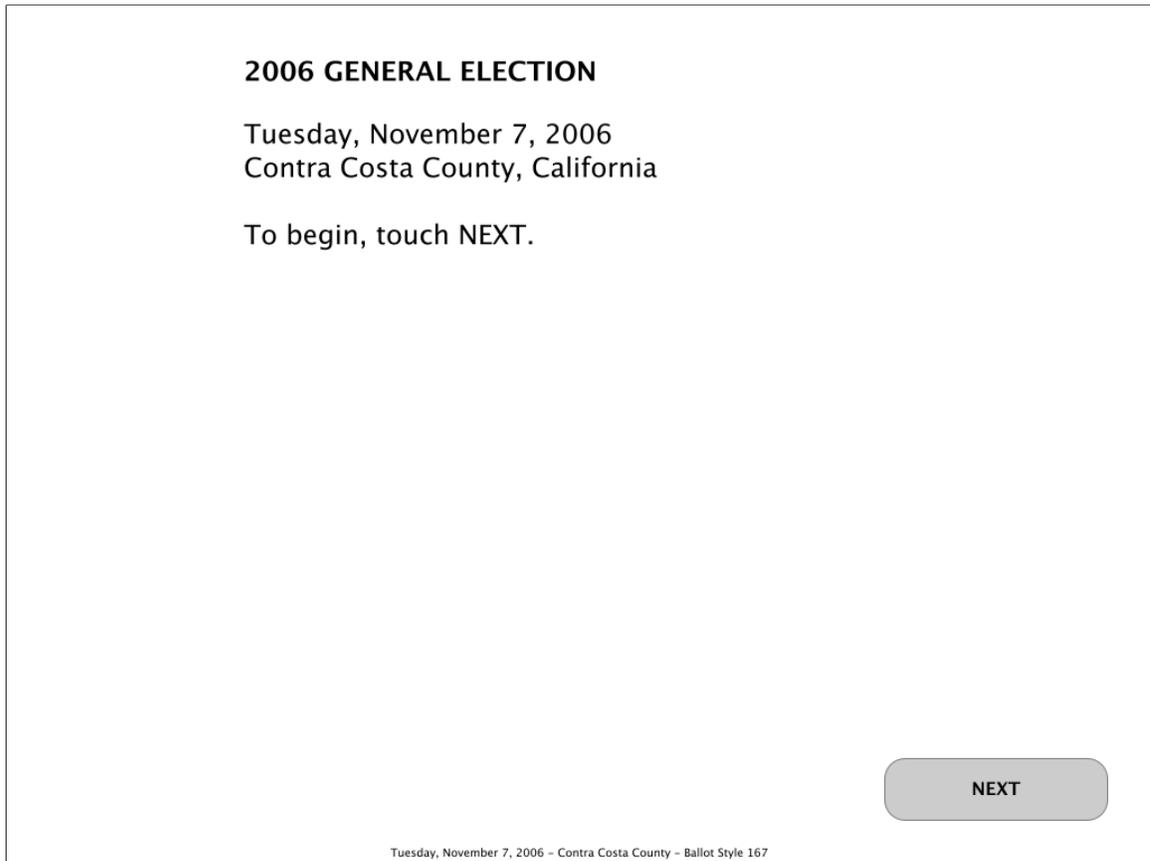
Group 8. This is the contest for Proposition 1A, with `max_sels = 1`, `max_chars = 0`, and `option_clips = 2`. It contains 2 options, with two sprites for each option:



Option 0 has two audio clips that both say “yes”; option 1 has two audio clips that both say “no”. (The redundant audio clips are unnecessary; this is just due to the current ballot compiler’s assumption that every option has a short and a long audio description.) Both options have `writeln_group = None`.

Group 9. This is the contest for Proposition 1B, with `max_sels = 1`, `max_chars = 0`, and `option_clips = 2`. It contains the same options as group 8.

Page 0. This is the screen image for layout 0.



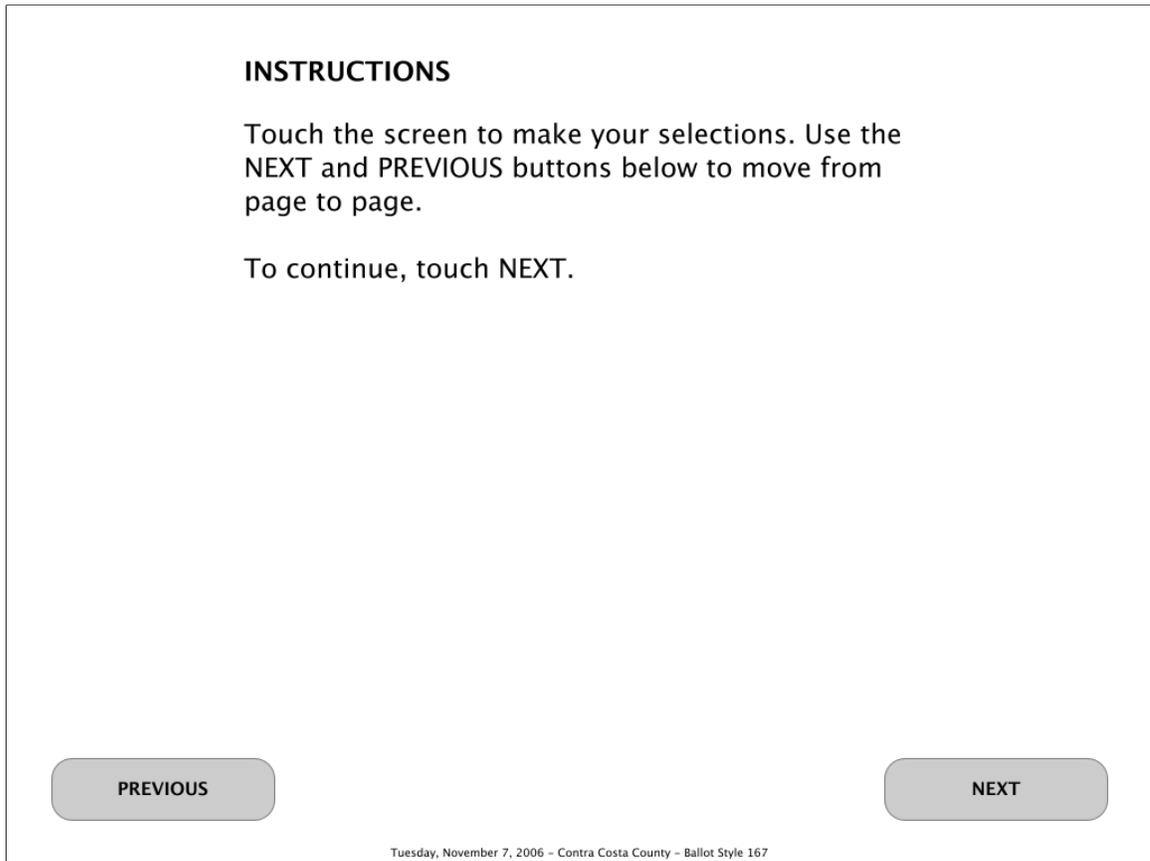
Page 0 has just one state, state 0, with the following audio message:

This is the General Election for Tuesday, November 7, 2006, Contra Costa County, California. To begin, touch NEXT in the lower-right corner of the screen. There is also a number keypad directly below the screen. The numbers are arranged like a telephone, with 1, 2, and 3 in the top row, 4, 5, and 6 in the second row, 7, 8, and 9 in the third row, and 0 in the bottom row. To begin, press 6.

There is a target positioned over the **NEXT** button; the **6** key and this target are both bound to a transition to page 1. (When no state is mentioned, state 0 is implied.)

Throughout the ballot, the arrangement of keypad controls is loosely associated with directional movement. The **4** and **6** keys (left and right) always navigate to the previous and next page; the **2** and **8** keys (up and down) navigate to the previous and next item on the page; and the **5** key (in the center) selects or activates the current item.

Page 1. This is the screen image for layout 1.

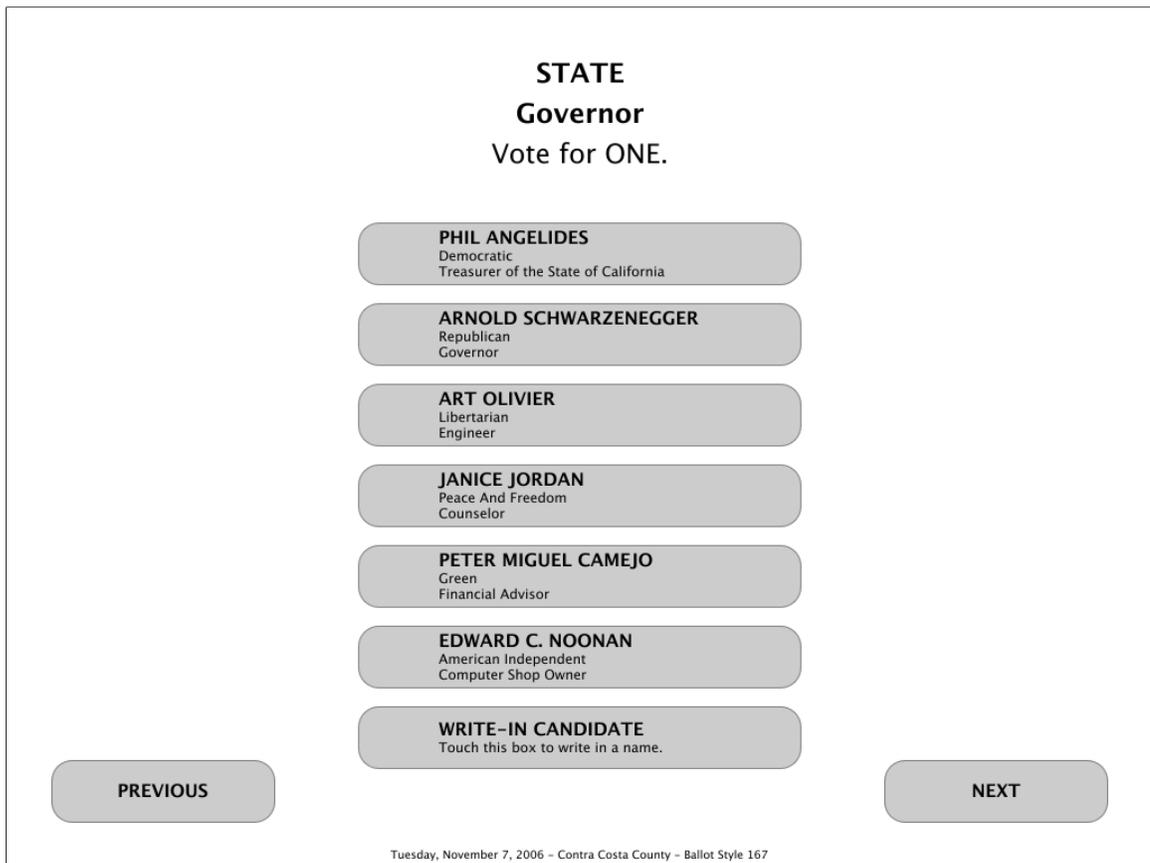


Page 1 has just one state, state 0, with the following audio message:

Touch the screen to make your selections. Use the NEXT and PREVIOUS buttons below to move from page to page. To continue, touch NEXT or press 6 on the number keypad.

There are targets positioned over the **PREVIOUS** and **NEXT** buttons. The **6** key and the **NEXT** target are bound to a transition to page 2. The **4** key and the **PREVIOUS** target are bound to a transition to page 0.

Page 2. This is the screen image for layout 2.



Page 2 demonstrates one possible way to present a single-selection contest. Touching any item changes the selection to that item, automatically deselecting any previous selection. The voter can also step through the options one by one, using the audio interface and keypad buttons. For voters who are using the visual and audio interfaces together, selecting an option by touchscreen also produces audio confirmation, and the options are also visually highlighted when the keypad buttons are used to step through them.

Page 2 has 8 states. State 0 has the following audio message:

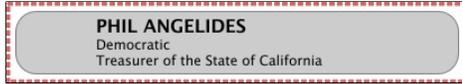
State. Governor. There are 6 candidates. Please vote for one.

No choices are currently selected. Your current selection is *list of selected options*.

Touch the screen to make selections or press 8 to hear the choices. To skip to the next contest, press 6.

The number of selections determines whether No choices... or Your current... is played.

In state 0, the **8** key is bound to a transition to state 1. States 1 through 7 correspond to the seven options for Governor. Each state highlights an option with a dotted red box. For example, state 1 places this sprite over the first option:



Each of the states 1 through 6 has an audio message of the form:

candidate name. This choice is currently selected. To select this choice, press 5.

To hear the next choice, press 8. To hear your current selections for Governor, press 3. To clear your selections for Governor, press 1.

This choice... or To select... is played depending on whether the option is selected. In these states, the **8** and **2** keys transition to the next and previous states. The **5** key clears groups 0 and 1, selects the highlighted option, and plays the audio message:

Selected *candidate name* for Governor.

State 7, in which the last option is highlighted, has the audio message:

Write-in candidate.

This choice is currently selected. To edit or cancel this write-in, press 5.

To write in a name, press 5. To hear all the choices again, press 4. To hear your current selections for governor, press 3. To clear your selections for governor, press 1.

This choice... or To write in... is played depending on whether the option is selected. In this state, the **5** key transitions to page 11, which is the write-in page for Governor.

Page 2 has 7 option areas, located over the 7 choices for governor. Each of the first six option areas has a corresponding target that clears groups 0 and 1 and then selects the option. There is a target positioned over the last option that transitions to page 11, which is the write-in entry page for Governor. The page also has a review area for group 1, with 25 small slots arranged in a row over the last option. This review area displays the entered text for the write-in candidate. When the write-in candidate has been selected, the highlighted sprite (with the check mark and green background) is pasted over the last option, and the review area causes the entered characters to be pasted on top of that.

There is a page-wide binding for the **1** key that clears groups 0 and 1 and plays the audio message:

The selections for Governor are now cleared.

There is also a page-wide binding for the **3** key that triggers the audio message:

Governor. No choices are currently selected. Your current selection is
list of selected options.

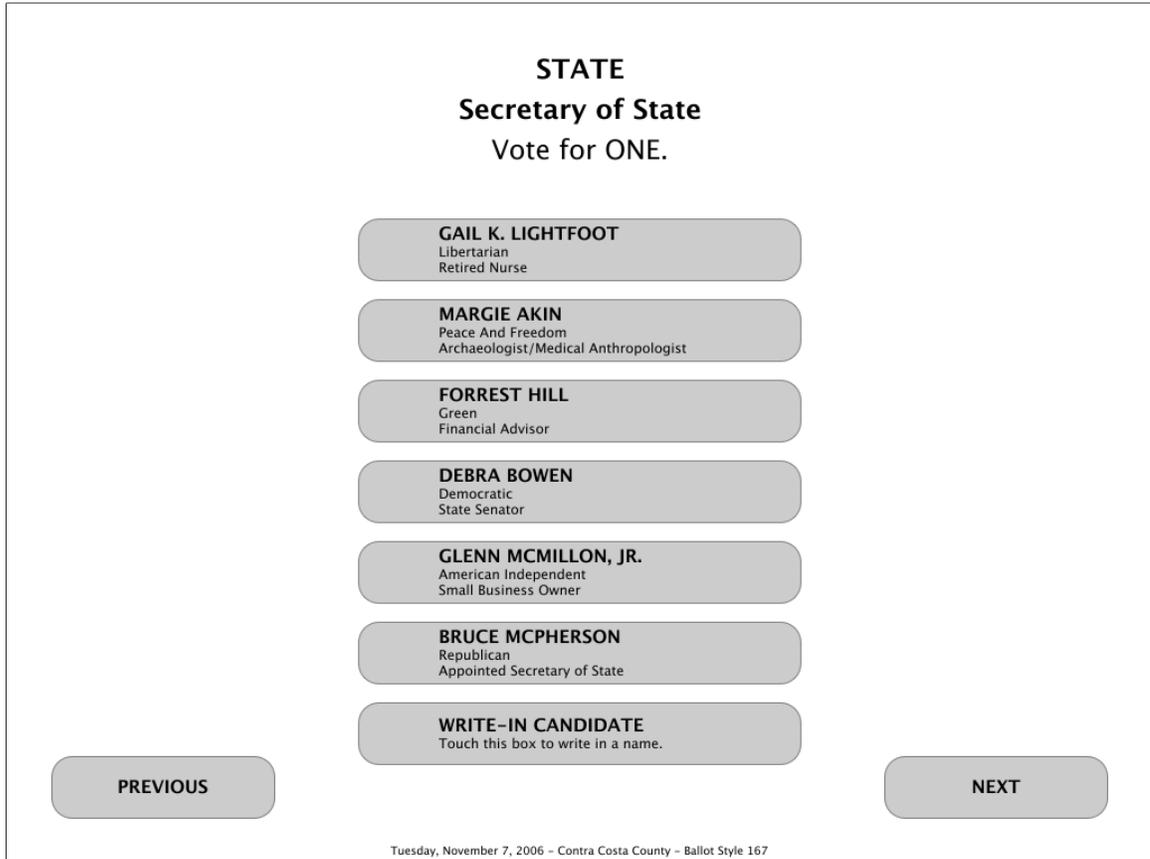
There are targets positioned over the **PREVIOUS** and **NEXT** buttons. The **6** key and the **NEXT** target are bound to a transition to page 3. The **4** key and the **PREVIOUS** target are bound to a transition to page 1.

The page also has one counter area, positioned over the **NEXT** button. This is a counter area for group 0, and its sprites look like this:



This counter area demonstrates one way of alerting voters when they proceed to the next contest without making a selection. When the number of selections is zero, the **NEXT** button is visually replaced with the **SKIP CONTEST** image; its behaviour is unchanged.

Page 3. This is the screen image for layout 3.



Page 3 has 8 states. State 0 has the following audio message:

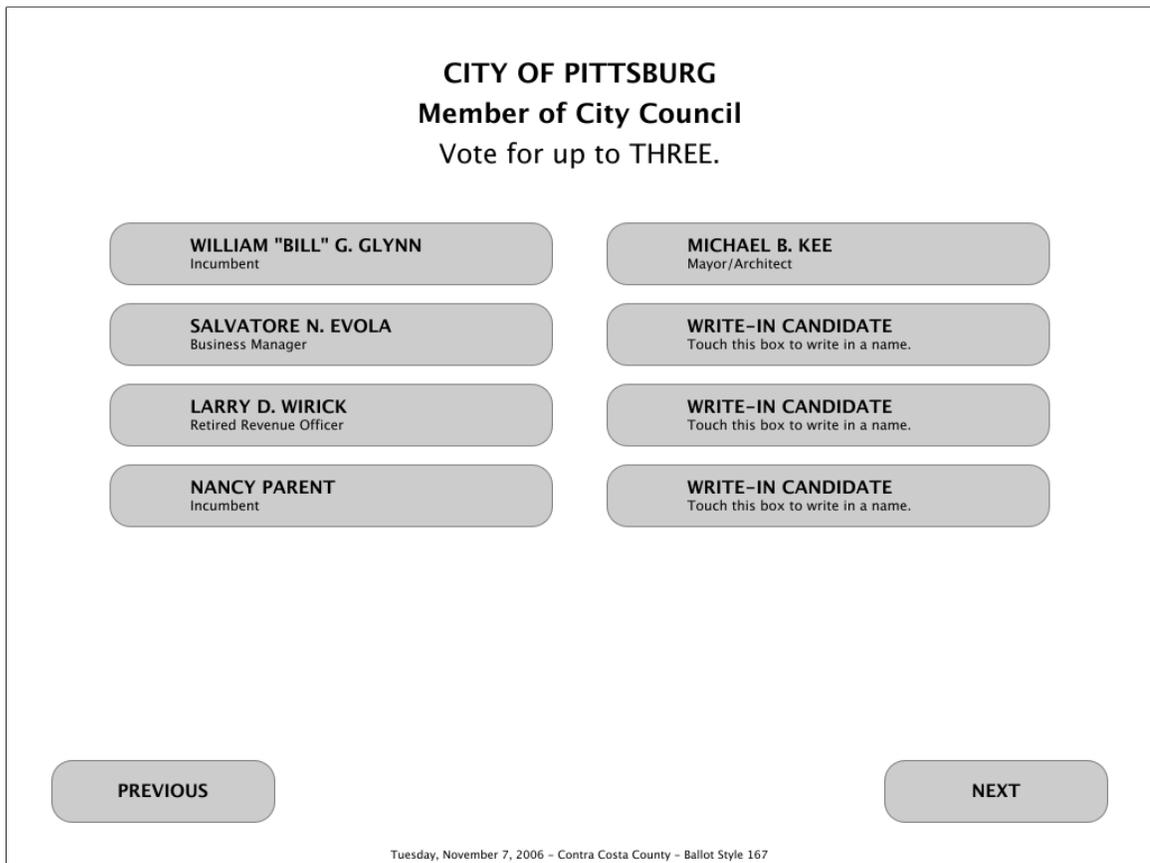
State. Secretary of State. There are 6 candidates. Please vote for one.

[No choices are currently selected.] Your current selection is [list of selected options].

Touch the screen to make selections or press 8 to hear the choices. To skip to the next contest, press 6. To go back to the previous contest, press 4.

The structure of the page is the same as page 2: states 1 through 7 highlight each of the options, and they have the similar bindings and audio messages to those on page 2. There are 7 option areas with corresponding targets that select them, and a review area for the write-in characters in group 3, positioned over the last option. Selecting the write-in option transitions to page 12, the write-in page for Secretary of State. There are targets positioned over the **PREVIOUS** and **NEXT** buttons, with a counter area over the **NEXT** button to replace it with a **SKIP CONTEST** image. The **6** key and the **NEXT** target go to page 4; the **4** key and the **PREVIOUS** target go to page 2.

Page 4. This is the screen image for layout 4.



Page 4 demonstrates a possible way of presenting a multiple-selection contest. Touching an option toggles whether it is selected or not, except that overvoting is prevented; attempting to overvote yields an audio explanation.

Page 4 has 9 states. State 0 has the following audio message:

City of Pittsburg. Member of City Council. There are 5 candidates. Please vote for up to 3. **No choices are currently selected.** **Your current selection is**—
Your current selections are *list of selected options*. Touch the screen to make selections or press 8 to hear the choices. To skip to the next contest, press 6. To go back to the previous contest, press 4.

The current number of selections determines which of the three clips are played: **No choices...**, **Your current selection is**, or **Your current selections are**. In state 0, the 8 key is bound to a transition to state 1.

States 1 through 8 correspond to the eight options. Because up to three selections are allowed in this contest, there are three write-in options. Each state highlights an option with a dotted red box, just like the pages for Governor and Secretary of State.

Each of the states 1 through 5 has an audio message of the form:

candidate name. To select this choice, press 5.—

This choice is currently selected. To deselect it, press 5.—

The maximum number of choices is currently selected. If you want to select more choices, you must first deselect a choice.

If you are done with this contest, press 6. To hear the next choice, press 8. To hear your current selections for Member of City Council, press 3. To clear your selections for Member of City Council, press 1.

To select... is played if the option is not selected and the group is not full; This choice... is played if the option is selected; and The maximum... is played if the option is not selected and the group is full. In these states, the 8 key goes to the next state and the 2 key goes to the preceding state. If the highlighted option is selected, the 5 key deselects it and plays the message:

Deselected *candidate name* for Member of City Council.

If the option isn't selected and the group is not full, the 5 key selects it and plays:

Selected *candidate name* for Member of City Council.

If the option isn't selected and the group is full, the 5 key plays the audio message:

You may only vote for up to 3 choices for Member of City Council. To vote for this choice, you must deselect another choice first. Your current selections are

list of selected options.

States 6, 7, and 8, which correspond to the write-in options, have the audio message:

Write-in candidate. To write in a name, press 5.—

This write-in is currently selected. To edit or cancel this write-in, press 5.—

The maximum number of choices is currently selected. If you want to select more choices, you must first deselect a choice.

If you are done with this contest, press 6. To hear the next choice, press 8. To hear your current selections for Member of City Council, press 3. To clear your selections for Member of City Council, press 1.

As with states 1 through 5, To write in... is played if the option is not selected and the group is not full; This choice... is played if the option is selected; and The maximum... is played otherwise. The **8** and **2** keys navigate between states. If the option is selected, or if it isn't selected and the group is not full, the **5** key jumps to the corresponding write-in page (page 13, 14, or 15). If the option isn't selected and the group is full, the **5** key produces the same message as in states 1 through 5:

You may only vote for up to 3 choices for Member of City Council. To vote for this choice, you must deselect another choice first. Your current selections are *list of selected options*.

Page 4 has 8 option areas, located over the 8 choices for City Council. Each of the option areas has a target with a page-wide binding just like the binding described above for the **5** key in states 1 through 8. The page has 3 review areas located over the last three options; these are for groups 5, 6, and 7, the write-in groups for this contest.

Just like pages 2 and 3, there are targets positioned over the **PREVIOUS** and **NEXT** buttons, with a counter area over the **NEXT** button to replace it with a **SKIP CONTEST** image. The **6** key and the **NEXT** button go to page 5; the **4** key and the **PREVIOUS** button go to page 3.

Page 5. This is the screen image for layout 5.

STATE MEASURES
Proposition 1A
Choose YES or NO.

TRANSPORTATION FUNDING PROTECTION. LEGISLATIVE CONSTITUTIONAL AMENDMENT.

Protects transportation funding for traffic congestion relief projects, safety improvements, and local streets and roads. Prohibits the state sales tax on motor vehicle fuels from being used for any purpose other than transportation improvements. Authorizes loans of these funds only in the case of severe state fiscal hardship. Requires loans of revenues from states sales tax on motor vehicle fuels to be fully repaid within the three years. Restricts loans to no more than twice in any 10-year period. Fiscal Impact: No revenue effect or cost effects. Increases stability of funding to transportation in 2007 and thereafter.

YES

NO

PREVIOUS

NEXT

Tuesday, November 7, 2006 - Contra Costa County - Ballot Style 167

Page 5 demonstrates one way to present a contest with a small, fixed number of choices. This example is a referendum with only two choices, so it's possible to map them directly to two buttons instead of highlighting each choice in a separate state. A non-touchscreen user can choose an option just by pressing the button for that option, instead of stepping through the options to find the desired one.

Page 5 has 3 states. State 0 has the following audio message:

State Measures. Proposition 1A. No choices are currently selected. Your current selection is *list of selected options*. To hear the full text of this proposition, press 8. Touch your selection on the screen, or, to select yes, press 7; to select no, press 9.

In state 0, the 8 key transitions to state 1.

State 1 has the audio message:

Transportation funding protection. Legislative constitutional amendment. Protects transportation funding... *text of paragraph describing proposition* ...in 2007 and thereafter. To hear the text of this proposition again, press 8. Touch your selection on the screen, or, to select yes, press 7; to select no, press 9.

In state 1, the **8** key transitions back to state 1, which causes the audio message to repeat.

There are two option areas on the page, one for **YES** and one for **NO**. There are two targets, one located over each option, and page-wide bindings for the **7** and **9** keys. The **7** key and the **YES** target clear the contest (group 9) and select option 0 for yes; the **9** key and the **NO** target clear the contest (group 9) and select option 1 for no. Both keys and both targets trigger the audio message:

Selected *option name* on Proposition 1A.

As on the preceding pages, there are targets positioned over the **PREVIOUS** and **NEXT** buttons, with a counter area over the **NEXT** button to replace it with a **SKIP CONTEST** image. The **6** key and the **NEXT** target go to page 6; the **4** key and the **PREVIOUS** target go to page 4.

Page 6. This is the screen image for layout 6.

STATE MEASURES
Proposition 1B
Choose YES or NO.

HIGHWAY SAFETY, TRAFFIC REDUCTION, AIR QUALITY, AND PORT SECURITY BOND ACT OF 2006.

This act makes safety improvements and repairs to state highways, upgrades freeways to reduce congestion, repairs local streets and roads, upgrades highways along major transportation corridors, improves seismic safety of local bridges, expands public transit, helps complete the state's network of car pool lanes, reduces air pollution, and improves anti-terrorism security at shipping ports by providing for a bond issue not to exceed nineteen billion nine hundred twenty-five million dollars (\$19,925,000,000). Fiscal Impact: State costs of approximately \$38.9 billion over 30 years to repay bonds. Additional unknown state and local operations and maintenance costs.

YES

NO

PREVIOUS

NEXT

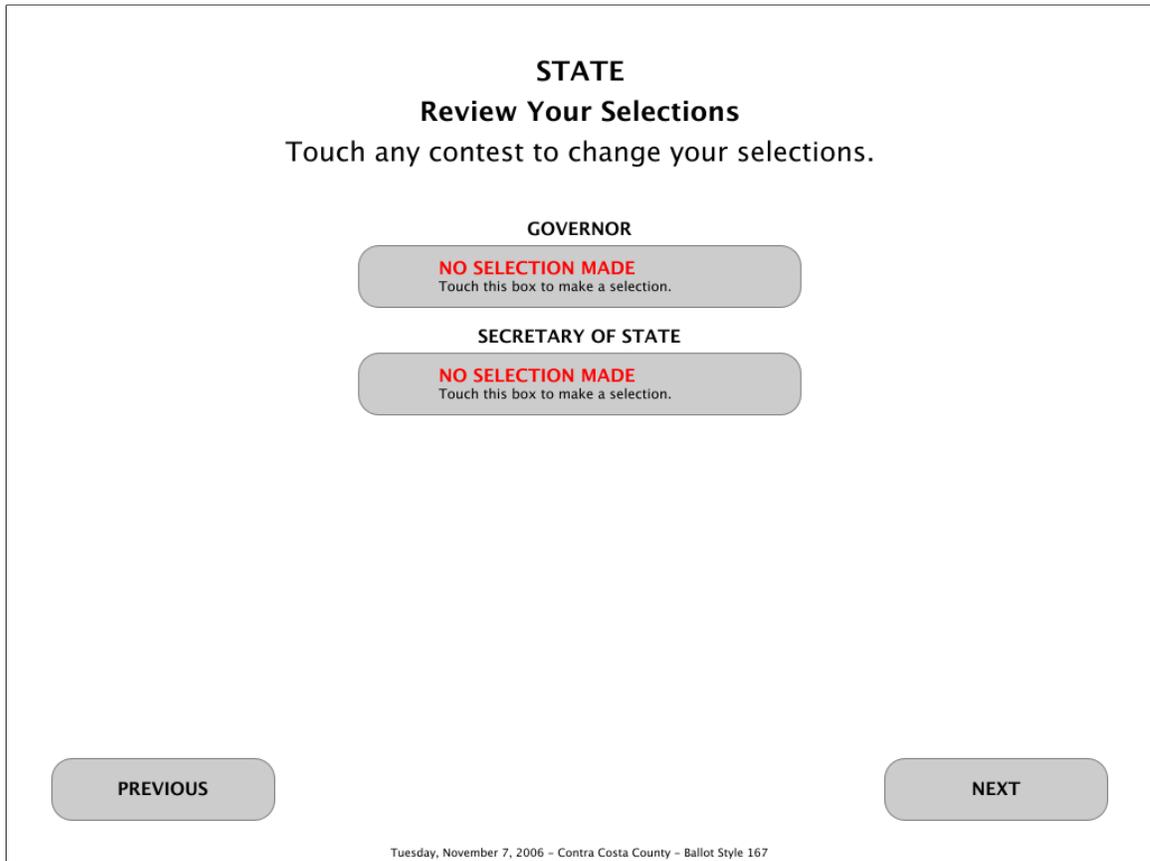
Tuesday, November 7, 2006 - Contra Costa County - Ballot Style 167

Page 6 has 2 states. State 0 has the following audio message:

State Measures. Proposition 1B. No choices are currently selected. Your current selection is *list of selected options*. To hear the full text of this proposition, press 8. Touch your selection on the screen, or, to select yes, press 7; to select no, press 9.

The structure of the page is the same as page 5: the 8 key transitions to state 1, with an audio message that reads out the text of the onscreen description. The 7 and 9 keys and YES and NO buttons work as on page 5. There are targets positioned over the PREVIOUS and NEXT buttons, with a counter area over the NEXT button to replace it with a SKIP CONTEST image. The 6 key and the NEXT target go to page 7; the 4 key and the PREVIOUS target go to page 5.

Page 7. This is the screen image for layout 7.



Pages 7, 8, and 9 allow the voter to review selections before casting the ballot. A voter using the audio interface can step through all the contests (automatically skipping from the end of one page to the beginning of the next) by repeatedly pressing the **8** key.

Page 7 has 3 states. State 0 has the following audio message:

Review your selections before casting your ballot. To change your selections for any contest, touch that contest on the screen. Use the NEXT and PREVIOUS buttons to move from page to page. Or, to hear your selections read back to you, press 8.

In state 0, the **8** key transitions to state 1, which has the audio message:

Governor. You have not made a selection for this contest. Your current selection is *list of selected options*. To make a selection To change your selection, press 5. For the next contest, press 8.

State 1 highlights the Governor contest with a dotted red box by placing this sprite over it:



In state 1, the **5** key transitions to state 0 of page 2, and the **8** key transitions to state 2 of page 7. State 2 has the audio message:

Secretary of State. You have not made a selection for this contest. —
Your current selection is *list of selected options*. To make a selection —
To change your selection, press 5. For the next contest, press 8. For the previous contest, press 2.

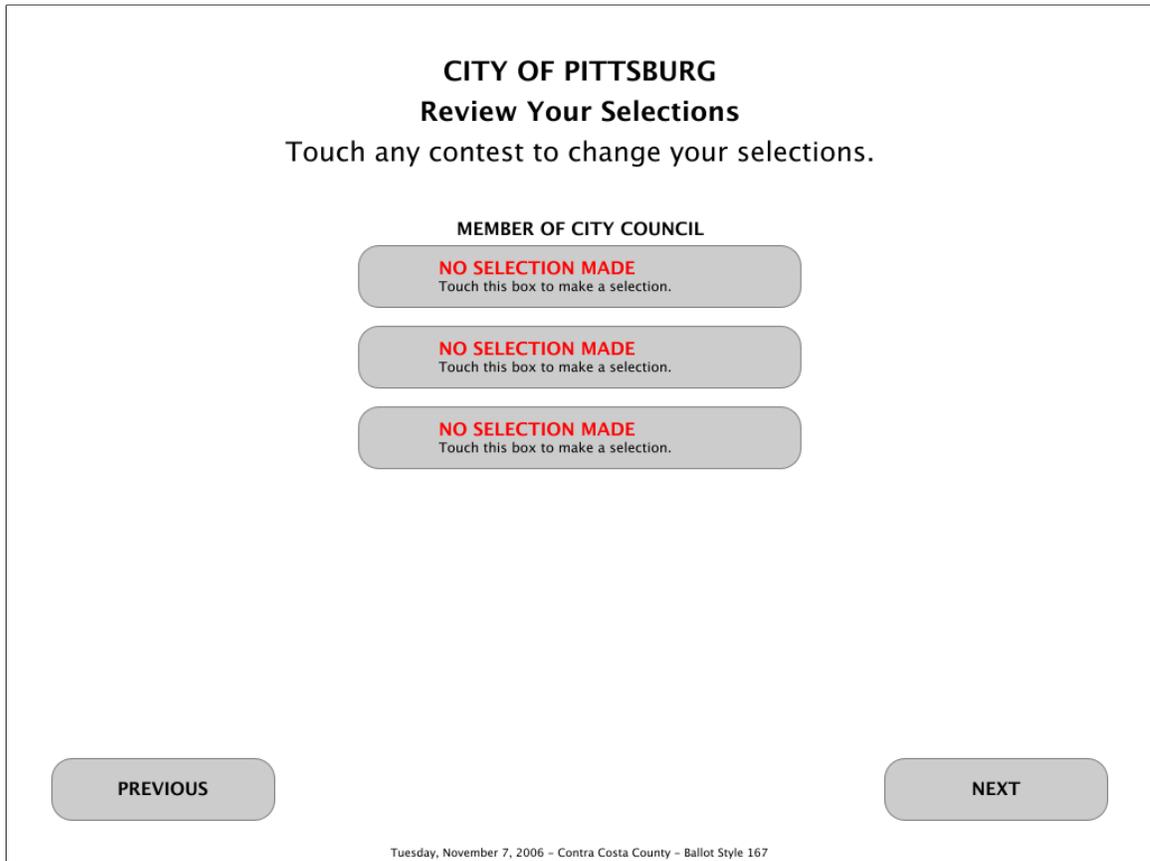
State 2 highlights the second contest with its sprite:



In state 2, the **5** key transitions to state 0 of page 3, the **8** key transitions to state 1 of page 8, and the **2** key transitions to state 1 of page 7.

The page has two review areas: one for group 0, positioned to overlay the box under “Governor”, and one for group 2, positioned to overlay the box under “Secretary of State.” Thus, when there is no selection, the **NO SELECTION MADE** message shows through from the background; when there is a selection, it covers up the **NO SELECTION MADE** message. There is a target positioned over each of the two contests; these targets transition to pages 2 and 3 respectively. There are also targets positioned over the **PREVIOUS** and **NEXT** buttons. The **6** key and the **NEXT** target go to page 8; the **4** key and the **PREVIOUS** target go to page 6.

Page 8. This is the screen image for layout 8.



Page 8 shows just one contest. (On a larger ballot, there could be many contests on each review page.)

Page 8 has 2 states. State 0 has the same audio message as page 7:

Review your selections before casting your ballot. To change your selections for any contest, touch that contest on the screen. Use the NEXT and PREVIOUS buttons to move from page to page. Or, to hear your selections read back to you, press 8.

In state 0, the 8 key transitions to state 1, which has the audio message:

Member of City Council. You have not made a selection for this contest. —
Your current selection is — Your current selections are *list of selected options*.
To make a selection — To change your selection, press 5. For the next contest, press 8. For the previous contest, press 2.

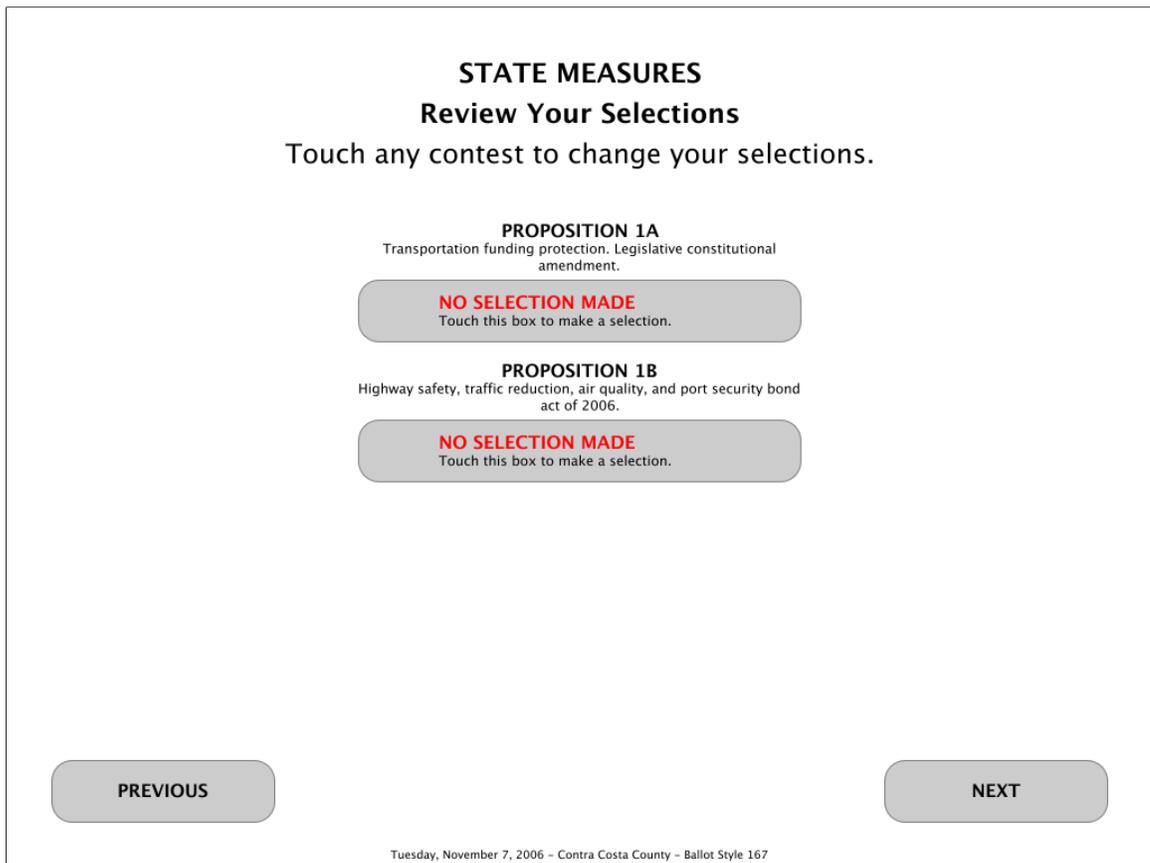
State 1 highlights the City Council contest with its sprite:



In state 1, the **5** key transitions to state 0 of page 4, the **8** key transitions to state 1 of page 9, and the **2** key transitions to state 2 of page 7.

The page has one review area for group 4, with its three slots positioned to overlay the three boxes under “Member of City Council.” When there are fewer than three selections in group 4, one of the **NO SELECTION MADE** messages will show through. There is one target positioned over this review area that transitions to page 4, as well as two targets positioned over the **PREVIOUS** and **NEXT** buttons. The **6** key and the **NEXT** target go to page 9; the **4** key and the **PREVIOUS** target go to page 7.

Page 9. This is the screen image for layout 9.

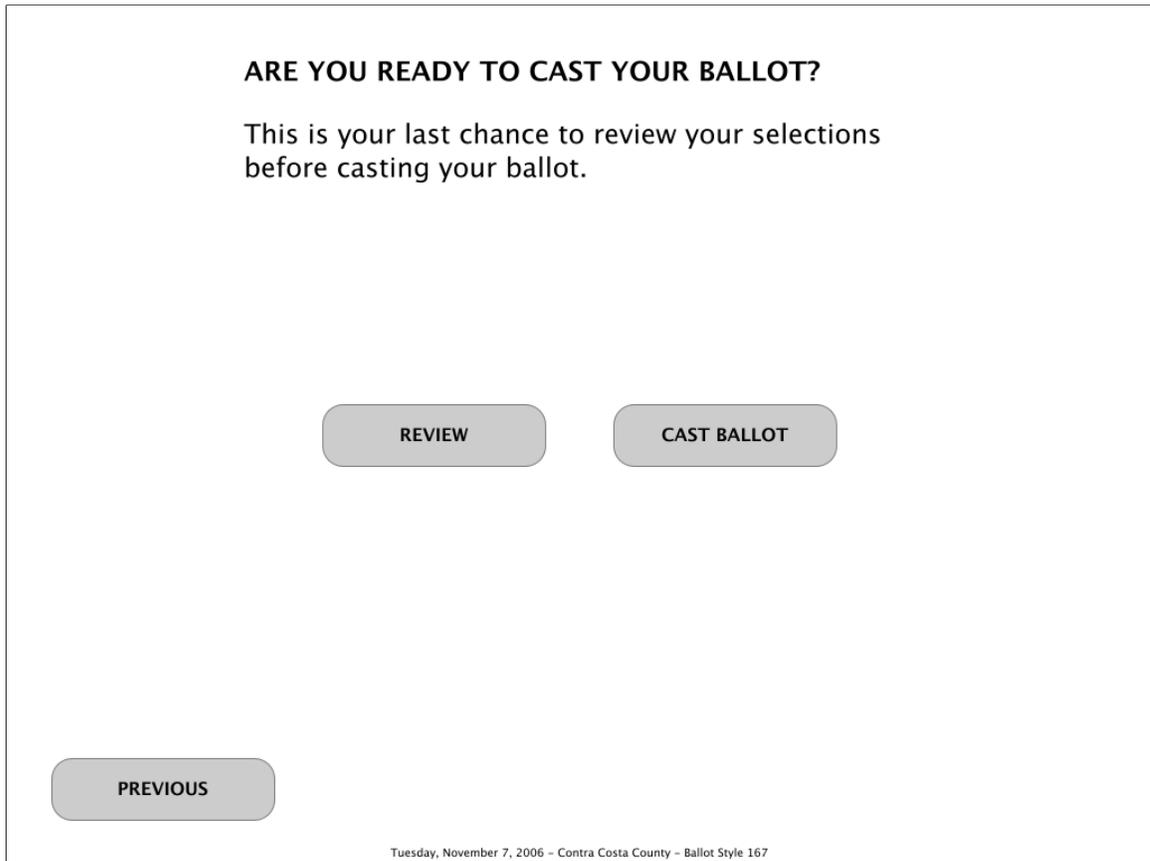


Page 9 has 3 states. State 0 has the same audio message as the previous two pages. States 1 and 2 correspond to the two propositions; each one highlights a proposition and reads back the selection for that proposition, similar to the previous two pages. In state 1, the **5** key transitions to state 0 of page 5, the **8** key transitions to state 2 of page 9, and the **2** key transition to state 1 of page 8. In state 2, the **5** key transitions to state 0 of page 6, the **2** key transitions to state 1 of page 9, and the **8** key produces the audio message:

This is the last contest. To proceed with casting your ballot, press 6.

The page has two review areas positioned over the two boxes for Propositions 1A and 1B, for group 8 and group 9 respectively, and targets over these regions that transition to page 5 and page 6 respectively. For the whole page, the **6** key and the **NEXT** target go to page 10; the **4** key and the **PREVIOUS** target go to page 8.

Page 10. This is the screen image for layout 10.



Page 10 is the final confirmation page before casting the ballot; it has just one state. State 0 has the audio message:

This is your last chance to review your selections before casting your ballot. To review your selections, press 1. To cast your ballot now, press 0.

The **1** key and the **REVIEW** button transition to page 7. The **0** key and the **CAST BALLOT** button transition to page 16. The **4** key and the **PREVIOUS** button transition to page 9.

Page 11. This is the screen image for layout 11.

WRITE-IN
Governor

Spell out the name using the letters below.

CLEAR BACKSPACE

A B C D E F G H I J
K L M N O P Q R S T
U V W X Y Z - ' SPACE

ACCEPT CANCEL

Tuesday, November 7, 2006 - Contra Costa County - Ballot Style 167

Pages 11 through 15 are pages for entering write-in candidates, corresponding to the write-in options in the Governor contest (1 write-in option), the Secretary of State contest (1 write-in option), and the City Council contest (3 write-in options). The voter can enter characters either by touching them on the screen or by using the keypad to step through the alphabet. The voter leaves the write-in page by either accepting or cancelling the write-in, which selects or deselects the corresponding write-in option.

Page 11 has 30 states. State 0 has the audio message:

Write-in candidate for Governor. This write-in is empty. This write-in contains *list of selected characters*. To write in a name, touch the letters on the screen. To edit this write-in, touch the letters on the screen. To delete the last letter, touch BACKSPACE or press 1.

Touch ACCEPT when you are finished, or touch CANCEL to cancel this write-in. Or, to advance through the alphabet using the keypad, press 6.

Whether the write-in is empty determines whether This write-in is empty. or This write-in contains is played, and also whether To write in a name... or To edit this write-in... is played. The **6** key advances to state 1.

State 1 highlights the **A** button with this sprite:



and has the audio message:

A. To add this letter to the write-in, press 5. To delete the last letter, press 1. To advance to the next letter of the alphabet, press 6. For the previous letter, press 4. To read back the letters you have entered, press 3. To accept this write-in, press 7. To cancel this write-in, press 9.

The name of the letter is spoken first so that the voter can quickly scan through the alphabet using the **6** and **4** keys to interrupt the message and navigate to the next and previous letters. The **7** and **9** keys express affirmative and negative actions, somewhat consistent with their use to select **YES** and **NO** on pages 5 and 6. the **1** key is used for deletion, somewhat consistent with its use to clear selections in other contests. And the **3** key is used to request a playback of selections, as it does on other pages.

States 2 through 29 highlight each of the other character buttons from **B** through **SPACE**, and they have similar audio messages. In all of these states, the **5** key appends the character to the group, the **1** key removes the last character, and the **6** and **4** keys transition to the next and previous state. In state 1, the **4** key goes to state 29; in state 29, the **6** key goes to state 1. If the group is not full, the **5** key appends the highlighted character to the group and plays the name of the character. If the group is full, the **5** key produces the audio message:

There is no room for more letters.

The page has one review area with 25 slots in a row over the green box at the top of the page. This review area shows the characters selected in group 1 and has a cursor sprite:



There are targets for each of the 29 letter buttons; each target is bound to the same action as the **5** key for that button (either it appends the character or notifies the voter that there is no more room).

There are targets over the **CLEAR** and **BACKSPACE** buttons. The **CLEAR** button clears the group and plays the audio message:

Clear.

If the group is empty, the **1** key and the **BACKSPACE** target just play the message:

This write-in is empty.

Otherwise, the **1** key and the **BACKSPACE** target remove the last character from the group.

There is a page-wide binding for the **3** key that plays the audio message:

This write-in is empty. This write-in contains *list of selected characters*.

There are also targets over the **ACCEPT** and **CANCEL** buttons. If the group is empty, the **7** key and the **ACCEPT** target just play the message:

This write-in is empty.

Otherwise, they clear group 0 (the contest for Governor) and select option 6 in group 0 (the write-in option for Governor), transition back to page 2, and play the message:

Selected write-in candidate *list of characters* for Governor.

The **9** key and the **CANCEL** target clear group 1 (this write-in group) remove option 6 from group 0 (the write-in option for Governor), transition back to page 2, and play the message:

Cancelled write-in.

Page 12. This is the screen image for layout 12.

WRITE-IN
Secretary of State
Spell out the name using the letters below.

CLEAR BACKSPACE

A B C D E F G H I J
K L M N O P Q R S T
U V W X Y Z - ' SPACE

ACCEPT CANCEL

Tuesday, November 7, 2006 - Contra Costa County - Ballot Style 167

Page 12 has 30 states, like page 11. State 0 has the audio message:

Write-in candidate for Secretary of State. This write-in is empty.

This write-in contains *list of selected characters*.

To write in a name, touch the letters on the screen.

To edit this write-in, touch the letters on the screen. To delete the last letter, touch BACKSPACE or press 1.

Touch ACCEPT when you are finished, or touch CANCEL to cancel this write-in. Or, to advance through the alphabet using the keypad, press 6.

The page has the same structure as page 11, except that it corresponds to group 3 (the write-in group for Secretary of State) and to option 6 of group 2 (the write-in option for Secretary of State), and transitions back to page 3 when the write-in is accepted or cancelled.

Page 13. This is the screen image for layout 13.

WRITE-IN
Member of City Council
Spell out the name using the letters below.

CLEAR BACKSPACE

A B C D E F G H I J
K L M N O P Q R S T
U V W X Y Z - ' SPACE

ACCEPT CANCEL

Tuesday, November 7, 2006 - Contra Costa County - Ballot Style 167

Page 13 has 30 states, like the other write-in pages. State 0 has the audio message:

Write-in candidate for Member of City Council. This write-in is empty.

This write-in contains *list of selected characters*.

To write in a name, touch the letters on the screen.

To edit this write-in, touch the letters on the screen. To delete the last letter, touch BACKSPACE or press 1.

Touch ACCEPT when you are finished, or touch CANCEL to cancel this write-in. Or, to advance through the alphabet using the keypad, press 6.

This page has the same structure as pages 11 and 12, except that it corresponds to group 5 (the first write-in group for Member of City Council) and to option 5 of group 4 (the first write-in option for Member of City Council), and transitions back to page 4 when the write-in is accepted or cancelled. When the write-in is accepted, group 4 is not cleared; option 5 is just added to the selections for group 4 since there can be multiple selections.

Page 14. This is the screen image for layout 14.

WRITE-IN
Member of City Council
Spell out the name using the letters below.



CLEAR BACKSPACE

A	B	C	D	E	F	G	H	I	J
K	L	M	N	O	P	Q	R	S	T
U	V	W	X	Y	Z	-	'	SPACE	

ACCEPT CANCEL

Tuesday, November 7, 2006 - Contra Costa County - Ballot Style 167

Page 14 is identical to page 13 except that it corresponds to group 6 (the second write-in group for Member of City Council) and to option 6 of group 4 (the second write-in option for Member of City Council).

Page 15. This is the screen image for layout 15.

WRITE-IN
Member of City Council
Spell out the name using the letters below.



CLEAR BACKSPACE

A	B	C	D	E	F	G	H	I	J
K	L	M	N	O	P	Q	R	S	T
U	V	W	X	Y	Z	-	'	SPACE	

ACCEPT CANCEL

Tuesday, November 7, 2006 - Contra Costa County - Ballot Style 167

Page 15 is identical to pages 13 and 14 except that it corresponds to group 7 (the third write-in group for Member of City Council) and to option 7 of group 4 (the third write-in option for Member of City Council).

Page 16. This is the screen image for layout 16.



Page 16 is the last page; transitioning here casts the ballot. There is just one state, and it has the audio message:

Thank you for voting. Your ballot has been recorded. *sound of applause*

There are no bindings on this page.

D Sample Pvote ballot designs

This appendix presents a few other possible designs for electronic ballots that could work with Pvote, to illustrate the flexibility of Pvote to handle other visual appearances and interaction styles.

An alternate visual design.

Proposition 1
All squirrels shall be banned from entering any campus of the University of California.

Choose ONE box.

YES NO

*Your ballot has **not yet been recorded.**
Press **NEXT** to continue or **FINISH** to review your selections.*

START OVER CLEAR FINISH NEXT

This is an example of a selection page with a different “look and feel” than the sample ballot in Appendix C. The video display has a different resolution (640 × 480 pixels), and the buttons appear shiny instead of flat. Square buttons are used for options and rounded buttons are used for navigation.

In Pvote, this design can be implemented just by drawing different full-screen images for each page and providing option sprites that match the new “look and feel.” For example, when the **YES** and **NO** options are selected, they can be overlaid with these sprites:



Random-access navigation.

Tuesday, November 7, 2006 — Contra Costa County — 167

Ballot Contents, part 1 of 2

INSTRUCTIONS
How to Make Selections
How to Cast Your Ballot

STATE

YOU ARE HERE → **Governor**

Lieutenant Governor	done
Secretary of State	no selection
Controller	no selection
Treasurer	done
Attorney General	done
Insurance Commissioner	no selection
Member, Board of Equalization	no selection

UNITED STATES CONGRESS

United States Senator	done
United States Representative	no selection

MEMBER OF THE STATE ASSEMBLY

11th District	no selection
---------------	--------------

JUDICIAL

Supreme Court	
Associate Justice, Seat 1	done
Associate Justice, Seat 2	done
Court of Appeals, 1st District, Div. 1	
Presiding Justice	done
Associate Justice, Seat 1	done
Court of Appeals, 1st District, Div. 2	
Associate Justice, Seat 1	done
Associate Justice, Seat 2	done
Court of Appeals, 1st District, Div. 3	
Presiding Justice	no selection
Associate Justice, Seat 1	done
Court of Appeals, 1st District, Div. 4	
Presiding Justice	done
Associate Justice, Seat 1	no selection
Associate Justice, Seat 2	done
Court of Appeals, 1st District, Div. 5	
Presiding Justice	done

▼ skip ahead to part 2 of 2 ▼

STATE
Governor
Vote for One

PHIL ANGELIDES
Democratic
Treasurer of the State of California

ARNOLD SCHWARZENEGGER
Republican
Governor

ART OLIVIER
Libertarian
Engineer

JANICE JORDAN
Peace And Freedom
Counselor

PETER MIGUEL CAMEJO
Green
Financial Advisor

EDWARD C. NOONAN
American Independent
Computer Shop Owner

WRITE-IN
Select this box to write in a candidate.

▲ **PREVIOUS** **NEXT** ▼ **CLEAR THIS QUESTION** **REVIEW AND CAST BALLOT**

This design offers a high-level overview of the ballot, always visible on the left third of the display. The overview region allows the voter to jump directly to any contest on the ballot, and also provides an indication of which contests are undervoted at all times. The right two-thirds of the display are similar to the ballot design in Appendix C.

In Pvote, this design can be implemented by including the overview pane with its **YOU ARE HERE** arrow as part of the full-screen image for each page. The undervote indicators next to each contest in the overview pane can be implemented with counter areas for each contest.

Persistent review.

This design is a variant of the previous random-access design. Instead of merely showing which contests are undervoted, the overview pane now shows the selection that the voter made. Thus, the overview pane functions as an everpresent review screen.

In Pvote, this design can be implemented by adding an “indicator group” to correspond to each contest group. Each indicator group would contain “indicator options” with small indicator-size sprites representing each option. Every operation that selects or deselects an option would also select or deselect the corresponding indicator option. Then the review indicators in the overview pane would be implemented as review areas for the indicator groups corresponding to each contest group.

The tediousness and redundancy in such a ballot definition suggests that Pvote could be improved by extending the ballot definition format to allow each option to be represented by an arbitrary number of sprites of different sizes, instead of just two sprites (selected and unselected) of the same size. Such an extension would also improve Pvote’s support for ballots that accommodate vision-impaired users (see page 104) or ballots that allow the voter to switch languages in mid-session.

Imitation paper ballot.

STATE ESTADO	STATE ESTADO	STATE ESTADO
<p>Governor Gobernador Vote for One Vote por Uno</p> <p><input type="radio"/> PHIL ANGELIDES Democratic Demócrata Treasurer of the State of California Tesorero del Estado de California</p> <p><input type="radio"/> ARNOLD SCHWARZENEGGER Republican Republicano Governor Gobernador</p> <p><input type="radio"/> ART OLIVIER Libertarian Libertario Engineer Ingeniero</p> <p><input type="radio"/> JANICE JORDAN Peace and Freedom Paz y Libertad Counselor Consejera</p> <p><input type="radio"/> PETER MIGUEL CAMEJO Green Verde Financial Advisor Consultor Financiero</p> <p><input type="radio"/> EDWARD C. NOONAN American Independent Independiente Americano Computer Shop Owner Proprietario de Empresa de Computadoras</p> <p><input type="radio"/> _____</p>	<p>Lieutenant Governor Vicegobernador Vote for One Vote por Uno</p> <p><input type="radio"/> LYNNETTE SHAW Libertarian Libertario Caregiver/Musician Cuidadora/Música</p> <p><input type="radio"/> JIM KING American Independent Independiente Americano Real Estate Broker Corredor de Bienes Raices</p> <p><input type="radio"/> JOHN GARAMENDI Democratic Demócrata California State Insurance Commissioner Comisionado de Seguros del Estado de Ca.</p> <p><input type="radio"/> TOM MCCLINTOCK Republican Republicano California State Senator Senador del Estado de California</p> <p><input type="radio"/> DONNA J. WARREN Green Verde Financial Manager/Author Administradora Financiera/Escritora</p> <p><input type="radio"/> STEWART A. ALEXANDER Peace and Freedom Paz y Libertad Automobile Sales Consultant Consultor de Ventas de Automóviles</p> <p><input type="radio"/> _____</p>	<p>Secretary of State Secretario de Estado Vote for One Vote por Uno</p> <p><input type="radio"/> GAIL K. LIGHTFOOT Libertarian Libertario Retired Nurse Enfermera Jubilada</p> <p><input type="radio"/> MARGIE AKIN Peace and Freedom Paz y Libertad Archaeologist/Medical Anthropologist Arqueóloga/Antropóloga Médica</p> <p><input type="radio"/> FORREST HILL Green Verde Financial Advisor Consultor Financiero</p> <p><input type="radio"/> DEBRA BOWEN Democratic Demócrata State Senator Senadora Estatal</p> <p><input type="radio"/> GLENN MCMILLON, JR. American Independent Independiente Americano Small Business Owner Propietario de Pequeña Empresa</p> <p><input type="radio"/> BRUCE MCPHERSON Republican Republicano Appointed Secretary of State Secretario de Estado Nombrado</p> <p><input type="radio"/> _____</p>
<p>← Page Página 01/12 →</p>		

This design emulates a paper ballot in its appearance and behaviour, offering a familiar interface for voters who are used to optically scanned ballots. The voter touches the candidates to fill in the bubbles and uses the arrow buttons at the bottom of the screen to flip through the ballot. Reviewing selections before casting the ballot consists of flipping back through the same pages and checking the marked bubbles, just as one would do with a paper ballot.

In Pvote, this design can be implemented by using empty and filled bubbles as the option sprites. The targets that select options can be large (covering the entire candidate name and description) while the corresponding option sprites are small (covering just the bubble).

E Pvote security review findings

This appendix presents the findings from the code review of Pvote, taken from the “Report on the Pvote security review” [93].

Correctness

The reviewers did not find any bugs in the original Pvote source code. However, they did find some errors and omissions in the assurance document.

Correctness claim for R1 (non-termination). Pvote is supposed to “never abort during a voting session” (R1). As part of the supporting argument for this claim, Section 7.11 of the assurance document describes how an upper bound on Pvote’s memory usage can be statically determined from the ballot definition. The memory usage argument identifies strings and lists as the only kinds of values with variable size, and establishes limits on how long they can possibly grow. But since Python (and Pthin) integers have unlimited range, a single integer can also have a variable size. The argument for R1 is incomplete because it neglects to establish any upper limit on the integer values used by Pvote.

However, the missing part of the argument can be filled in by examining all the expressions in the Pvote code that yield new integers. There are only four built-in functions that return integers, and all of them return values that are known to be bounded:

- `range()` yields a list of integers between 0 and its argument.
- `ord()` yields an integer between 0 and 255.
- `len()` yields the length of the list or string argument, and the argument in the assurance document already establishes that lists and strings have bounded size.
- `enumerate()` yields lists containing integers all between 0 and the length of the list, and the argument in the assurance document already establishes that list lengths are bounded.

Aside from built-in functions, the only other way to produce a new integer value is by performing arithmetic. Arithmetic expressions occur in the Pvote source code on the following lines:

- `Ballot.py` line 125: This line always yields an integer less than 2^{31} .
- `verifier.py` lines 23, 27, 31, 34; `Navigator.py` lines 28, 32, 46: These lines all increment an integer loop index by a constant or a quantity fixed in the ballot definition. The iteration count in each of these loops is determined by a fixed value in the ballot definition.
- `main.py` line 3; `verifier.py` lines 40, 42, 60, 64, 65, 67, 88; `Navigator.py` lines 12, 13, 27, 31; `Video.py` lines 20, 21; `Printer.py` line 12; `Audio.py` lines 28, 29, 35: These lines all perform arithmetic and do not store the result. The operands to the arithmetic expressions are all bounded values (constants, Boolean values such as 0 or 1, values fixed in the ballot definition, list lengths, or string lengths).
- `Navigator.py` lines 107, 109, 111: These lines perform arithmetic and pass the result to the `Audio.play()` method. The operands to the arithmetic expressions are all bounded values. The audio driver stores the clip indices, but does not perform any arithmetic on them.
- `Audio.py` line 22: This line performs arithmetic on `rate`, which is fixed in the ballot definition, and passes it to `put_int()`, which converts it to a string without storing it.
- `Audio.py` line 34: This line increments the stored integer `self.pos` by a passed-in value. In order for this integer to remain bounded, Pvote relies on Pygame's `Sound` constructor to stop calling `read()` after it returns an empty string to signal that the end of the file has been reached.

Correctness claim for R9 (ballot casting). Pvote is supposed to “commit the ballot when and only when so requested by the voter” (R9). By design, a Pvote ballot definition can specify a page transition to occur automatically after some amount of time has passed with no response from the user. Because a transition to the last page commits the ballot, this automatic timeout transition can be made to commit the ballot without explicit voter action, in violation of R9. A timeout transition could also prevent the user from committing by jumping to a

page with no escape; or it could indirectly force the user to commit by jumping to a page with no escape except to cast the ballot (the user has no way to go back and change selections).

Pvote's design assumes that the ballot definition file will be checked before an election (A5). Pvote should ensure that the ballot file will not cause Pvote to crash; the pre-election checks should ensure that the ballot does not mislead or misrepresent the voter. To uphold R9, one of these checks must ensure that no timeout transition deprives the user of the ability to cast the ballot or the ability to change their selections before casting the ballot. The assurance document failed to mention that such a check is necessary.

Missing requirement for voter privacy. The assurance document states no explicit requirement for preserving a voter's privacy once the voter's ballot has been committed. Although Pvote is restarted afresh for each new voter (A3), there is no assurance of privacy for the interval from when the voter walks away until the machine is reset. For example, a ballot definition with a review area on the last page might reveal the voter's choices to the pollworker or the next voter, without violating any requirements stated in the assurance document. There needs to be an assurance argument or a ballot definition audit requirement to ensure that the images and audio shown on the final page are independent of all prior choices. In combination with R3 (Pvote should become inert after a ballot is committed), this would ensure that the voter's choices will not be revealed after the voter commits the ballot.

Negative integers. The assurance document (in Section 7.1) makes an argument that negative integers are never used in Pvote. This argument claims to list all the uses of the subtraction operator in Pvote, but neglects to mention the expression `len(self.model.pages) - 1`, which appears on lines 12 and 13 of `Navigator.py`. Nonetheless, the claim that negative integers are never used still holds, since the verifier ensures that `model.pages` always has a length of at least 1.

Pthin specification. Pthin was intended to be a subset of Python in that any valid Pthin program is a valid Python program with the same behaviour. However, the Pthin specification does not accurately describe how a Pthin program would behave when run under a Python interpreter.

In some cases where Pthin specifies that a fatal error should occur, Python will not raise an exception. This is significant for Pvote because Pvote relies on fatal errors to ensure that invalid ballot definitions never make it past the verifier.

1. According to the Pthin specification, substring slicing `s[i:j]` should cause a fatal error unless $0 \leq i \leq j < n$, where n is the length of `s`, but Python actually accepts any integers for the starting and stopping indices.
2. According to the Pthin specification, list indexing `l[i]` should cause a fatal error unless $0 \leq i < n$, where n is the length of the list, but Python actually allows $-n \leq i < n$. The same holds for string indexing as well.
3. According to the Pthin specification, any type violation or illegal argument to a built-in operation causes a fatal error. But, if Pvote were to pass a callback function to Pygame, and that function were to throw an exception inside Pygame, then Pygame could catch the exception and thereby deviate from the Pthin specification.

The Pthin specification also deviates from the behaviour of the Python interpreter in the following ways:

4. The Pthin specification neglects to mention that `and` and `or` have short-circuit evaluation, as in Python.
5. The Pthin specification documents the `pop()` method with no arguments, but doesn't document `pop()` with one argument, which is used on line 16 of `Audio.py`.

Although the Pthin specification is in error, it does not appear that any of the above five deviations would cause Pvote to function incorrectly:

1. The verifier does not use the slicing operator, so there is no risk that the slicing operator will fail to produce a fatal error when it should.

2. Section 7.1 of the assurance document establishes that a negative integer never appears as a string or list index.
3. Pvote never passes any callback functions to Pygame.
4. The `and` and `or` operators are used at `main.py` line 24, `verifier.py` line 96, `Ballot.py` line 126, `Navigator.py` lines 12, 44, 50, 54, 80, 83, and 85, and `Video.py` lines 20 and 21. None of the operands cause side effects; among all these expressions, the only function calls are to the **`Navigator.test()`** method, and this method has no side effects.
5. This is simply a documentation error; no security claims rely on it.

Figure 6.1. A causal connection is missing from the diagram in Figure 6.1 of the assurance document. There should be a dotted line leaving the event loop to indicate that it schedules timer events, and another dotted line entering the event loop for the timer events it receives.

Consensus recommendations

This section describes recommendations made by reviewers on ways that Pvote or its assurance document could be improved to make Pvote easier to deploy, use, or review.

Assurance document. The reviewers agreed that the document should give a detailed breakdown of all the properties that need to be verified about a ballot definition, in three categories: those checked by human review, those checked by automated tools outside of Pvote, and those checked by Pvote's verifier.

The reviewers recommended that a section of the document should separately enumerate all causal connectivity with the outside world (e.g., primitives or library calls that have external effects, such as the `print` statement or the `open()` function).

The reviewers suggested that the assurance document should explicitly state, on line 89 of `Navigator.py`, the precondition that `audio.playing` has to be false by that point, and that if the program reaches this point, it has been false for at least the last `ballot.model.timeout_ms` milliseconds.

The reviewers recommended that the assurance document explicitly state that cursor sprites need to be checked to make sure they are not confusable with a candidate or a character.

The reviewers noted that Python dumps a stack trace when an exception is thrown. If an exception occurs during a voting session, a record of the corresponding stack trace could conceivably violate voter privacy. The reviewers recommended that the assurance document mention this issue and propose ways to deal with it.

Pthin. The reviewers recommended that the Pthin specification should prohibit all unprintable characters in source code except newline, and specifically should prohibit tab characters to avoid ambiguity in indentation levels. (It was confirmed that the Pvote source code contains no unprintable characters except newline.)

The reviewers recommended that Pthin should prohibit all

identifiers containing double-underscores except `__init__`, to avoid the possibility of triggering any special or implicit behaviours in Python.

The reviewers suggested that Pthin explicitly forbid nested class definitions and function definitions, for simplicity.

The reviewers suggested that Pthin could avoid some bugs caused by one-character changes from `==` to `=` by excluding chained assignments of the form `x = y = z`.

Ballot definition. The reviewers recommended that ballot definition analysis tools should be distributed with Pvote to help reviewers check commonly desired properties of ballot definitions. Some examples of such properties are reachability of all pages from the starting state, reachability of the commit page from any page, and reachability of all the selection pages from any page.

The reviewers suggested that the ballot definition's `int` type be renamed `nat` to make it more clear that this type excludes negative numbers.

The reviewers suggested that ballot definitions be digitally signed and that Pvote check the signature. The reviewers also agreed that the ballot definition file's 8-byte header should be included in the computation of the hash at the end of the file.

Serialization format. Some reviewers, concerned that the binary format of the ballot definition file would make it difficult for humans to examine, initially suggested XML as an alternative serialization format, with images and audio stored in auxiliary files. Other reviewers objected that XML is also unreadable. The reviewers reached the consensus that the ballot definition should remain the current binary format, so that the Pvote code for reading it can remain simple and elegant; a separate, textual ballot definition format should be specified so that the textual form can be put in a one-to-one correspondence with the binary form. The Pvote system should include a disassembler (that converts the binary form into the textual form together with any auxiliary binary files) and an assembler (that does the opposite).

No one has the option to write their own voting software and vote on it, but anyone who wants to verify a correct conversion has the option to write their own assembler and disassembler.

The reviewers thought it would also be nice to have a one-way translator that produces interactive HTML pages or a Flash animation, so that voters can visit a web page and preview the voting experience in a browser.

Implementation. The changes suggested by the reviewers are described here and also noted in the code listing in Appendix B.

Navigator. The reviewers agreed that the navigator should have something like a `self.committed` flag to indicate that the ballot has been committed, together with a `commit()` method that commits the ballot and sets the `committed` flag.

The reviewers felt that some method names in the navigator could be clarified, such as `press()`, `touch()`, `invoke()`, `execute()`.

The reviewers felt that lines 66 to 67 of `Navigator.py` were just “too clever for its own good.” The intent of these lines could be expressed more clearly by writing:

```
if cond.invert:
    result = not result
if not result:
    return 0
```

to show that `cond.invert` reverses the sense of the condition and that 0 is returned the only when the condition is not met.

The reviewers agreed that line 80 of `Navigator.py` could use some parentheses to clarify the Boolean expression.

The reviewers suggested eliminating the recursion in `review()` by duplicating the body of the method in two specialized methods, `review_contest()` and `review_writein()`. `review_contest()` would call `review_writein()` and there would be no recursive calls, making it easier for reviewers to understand.

The reviewers found **Navigator.execute()** more confusing than necessary because it uses both the list `self.selections` and a local variable `selections` that aliases a part of it. Mixing these two ways of accessing the list makes it harder to reason about the code, because each could have side-effects on the other. The method would be easier to verify if it always accessed the list through just `self.selections` or just `selections`.

Some reviewers were uncomfortable with the **get_option()** method, whose parameter is not limited to a specific type; it accepts any object with members named `group_i` and `option_i` (thus, any **Condition**, **Step**, or **Segment**).

Ballot. The reviewers suggested that the **Ballot** module would be easier to understand if the hashing were performed by a separate object, not the **Ballot** itself. This would also prevent other objects from having access to the incompletely constructed **Ballot** object during construction.

Verifier. The reviewers suggested that the verifier have separate methods **get_bool()** and **get_enum()** instead of **get_enum()** for both purposes, and separate methods **get_int()** and **get_intn()** instead of **get_int()** for both purposes.

The reviewers suggested that **get_str()** would be clearer if it checked `isprint(ch)` and `ch != '~'` rather than `32 <= ord(ch) <= 125`.

General style. The reviewers suggested that all the constants be moved to a single module and that each enumerated type be defined as a class that consolidates the cardinality of the enumeration, the symbolic names of the elements, and the values of the elements. The reviewers noted that, for example, **AUDIO_DONE** is assigned in two separate files, with no condition that they be assigned the same value.

The reviewers suggested that explicit `return None` statements be inserted where `None` is an intentionally returned value, instead of relying on `None` to be returned by default.

Inconclusive recommendations

This section contains recommendations made during the review that did not reach general agreement, were disputed, or were ultimately withdrawn.

Ballot definition. Some reviewers were concerned that each write-in option needs its own separate write-in text entry page, with the text entry state machine duplicated on each page. Thus, for example, for a ballot with two single-selection contests and two three-selection contests, if all the contests allow write-ins (in English letters), there will be eight nearly identical write-in pages with about 30 states each. This is because the VM doesn't have a stack, doesn't support subroutines, and can't pass parameters. It was suggested that ballot definition complexity could be substantially reduced by turning the VM from a finite-state machine to a pushdown automaton. Call-return semantics would also be useful not only for write-ins, but also for displaying help pages and revisiting contests from a review screen.

Other reviewers were not convinced that this duplication was that important. They felt that 30 states was not enough of an explosion of states to justify additional complexity in the ballot definition language. Ultimately there was no consensus that call-return should be added.

A possible compromise might be to create a deterministic compiler that translates from a language with a call-return feature to the current language without call-return, and then publish its input and output for verification.

Image format. Adding an alpha channel to images was suggested as a way of increasing flexibility in the design of the ballot definition. However, this would add a little more code to the voting machine and make human review of ballot definitions harder. The true appearance of the ballot might be hidden from human reviewers using alpha compositing tricks

(for instance, a sprite with an alpha channel could appear normal over one background but contain a hidden message that appears when it is composited over another background).

Programming language. Some reviewers objected to the use of chained-inequality expressions such as $x == y > z$ because they were potentially misleading for a reader used to the C interpretation; they recommended that this syntactic shorthand be removed from the Pthin specification and that the clauses be written out separately as $x == y$ and $y > z$. Others found such expressions sensible and concise.

Observations

This section documents other notable observations that reviewers made.

Single source vs. multiple sources. The reviewers agreed that the most critical code is code that:

- has to be in the voting machine,
- has to be correct, and
- cannot be multiply sourced.

Pthin. Some reviewers noted Pthin’s simplicity and readability, and mentioned that they were impressed at their ability to read and understand a language they didn’t know.

The definition of Pthin implies that a Pthin program has no access to information about its environment other than explicit user inputs, and therefore no way to distinguish a real election from a test. The assurance document could state explicitly that the Pthin language is deterministic and that it has no implementation-dependent or compiler-dependent features other than memory capacity limits (which, if exceeded, can only cause fatal errors).

The definition of Pthin helps support some of the assurance requirements:

- R5 says that Pvote’s behaviour in each session should be independent of any previous sessions. Satisfying this requirement doesn’t depend on the code of Pvote; it relies upon Pthin’s definition (e.g., no arbitrary access to the filesystem), together with the design choice that the pollworker resets the voting station.
- R7 says that Pvote’s behaviour should be determined entirely by the ballot definition and the stream of user input events. This also doesn’t depend on the code of Pvote; it is ensured by the interfaces to Pvote and the fact that Pthin is deterministic. Neither Pthin nor Pygame provide any access to clocks or sources of randomness.

Terminology. The definition of Pthin misuses the term “precondition.” A precondition is something that is assumed to be true, and if the precondition is violated then the resulting behaviour is undefined. However, in the Pthin definition, the word “precondition” is used to describe any condition whose violation is *required* to cause a fatal error. This distinction is important because such fatal errors are necessary to the assurance arguments that are made in the annotations on the Pvote source code.

Separation of concerns. The separation between the video driver and the navigator is a separation of space and time: the video driver knows about space but has no concept of time (no history); the navigator knows about time but knows nothing of space (screen layout).

A claim worth stating and verifying is that once the video driver receives a **goto** message, it should be history-insensitive about all prior state, as if a new video driver was freshly instantiated on each page transition.

Temporal categories of variables. One reviewer noted that many variables are intended to describe the state of the world at a particular time, either past, future, or present. For example, the navigator uses `self.page_i` to refer to the current page and the parameter `page_i` refers to what will become the current page. It would be helpful to have a naming convention to reflect this, so it is easy to tell what point in time a variable refers to. For example, the parameter `page_i` could be named `new_page_i` or `next_page_i`.

Something similar may also be useful in the audio module, which has to distinguish between what Pvote thinks the audio state is (busy or available) and what the Pygame audio driver thinks the audio state is.

Printer output. Some reviewers found the printer output unfriendly for human readers; in particular, they felt the insertion of markers after each write-in character was ugly.

Arithmetic. Some reviewers commented that arithmetic is difficult to reason about—it's something humans are especially bad at, compared to computers. In particular, the **Navigator.review()** method was harder to verify than it could have been, because it relies on arithmetic to establish a correspondence between the array of slots and various other structures. The reviewers found the incrementing of `slot_i` and the passing of `slot_i` recursively to **review()** tricky to understand (and hence suspicious).

Design consistency. The reviewers noticed that certain features of Pvote violated the design heuristic of prioritizing the simplicity of the ballot format:

- The `SG_MAX_SELS` audio segment type is not strictly necessary. Since the maximum number of selections in each contest is statically known, every instance of `SG_MAX_SELS` could be replaced by `SG_CLIP`. The ballot definition might be slightly harder to audit as a result.
- States are also not strictly necessary and could be eliminated. Each state could be turned into a separate page, at the cost of duplicating all the common information that states currently share.

Fleeing voters. Some local policies require that fleeing voters should have their ballots automatically cast for them. One way to implement this for Pvote would be to provide a special button on the machine (perhaps behind a locked door) that pollworkers could press to cast the ballot of a fleeing voter.

Code annotations. The assurance document presented a precondition/postcondition analysis as a set of annotations to the source code. This analysis was extremely tedious to perform by hand, even for less than 500 lines of code, and would also be extremely tedious for reviewers to verify by hand. The reviewers were concerned that annotations kept separate from the code would be difficult to maintain, and would be better expressed directly in the source code. The reviewers felt

that, to be practical, verification support based on annotations has to be cheap and has to require few annotations to be added by the programmer.

In a statically typed language, many or most of the annotations in the assurance document would have been unnecessary, and would be automatically checked by a compiler. In many reviewers' opinion, this affirmed the value of type systems for secure and reliable code.

Open issues

This section describes other unresolved issues and ideas that were discussed at the review concerning Pvote or software auditing in general.

Ballot definition. We discussed the following topics concerning the ballot definition.

Validity. How much should Pvote constrain the ballot definition? There is a trade-off between the strictness of the constraints enforced by Pvote's verifier and the length of time that the Pvote software goes unchanged between revisions. With too many constraints, we run the risk that unanticipated changes in laws and regulations (or differences in regulations among jurisdictions) may invalidate Pvote's assumptions and force Pvote to change frequently; this would argue for minimizing these constraints. New laws could also require Pvote to support new features, which similarly could require less constrained ballot definitions. On the other hand, too few constraints on the ballot definition would make it harder to ensure that Pvote doesn't crash.

There is also a trade-off between the ease of auditing a published ballot definition file and the size of the TCB. A higher-level ballot definition is easier for humans to audit, but is also likely to mean more code in Pvote.

Auditing. Instead of reviewing the ballot definition directly, assurance could be gained by publishing the input to the ballot layout tool and the code of the ballot layout tool. If the ballot layout tool is deterministic, anyone should be able to run it to regenerate the ballot definition file.

For auditing the ballot definition, it could also be helpful to be able to start from the ballot definition file and unambiguously recover the original input to the ballot layout tool (for example, by performing OCR on the images, perhaps

with some hints from the ballot layout tool). This might be a requirement to impose on the ballot layout tool.

Programming language. The effect of programming language design on source code review was another prominent topic.

Mistyped or confusing identifiers. Python automatically creates a new binding when you make a local assignment; thus, assigning to a misspelled variable name will just silently create another variable. The same is true for assignment to member variables. The reviewers considered this error-prone and suggested some ways to address the problem:

- Use a tool to check identifiers that are suspiciously similar.
- Use a tool to check for variables that are assigned but then unused.
- Require all functions to declare their local variables in comments or decorators and statically check these declarations.
- Require constructors to initialize all member variables, and forbid `self` from escaping the constructor before all fields are assigned.

One way for code to be (inadvertently or intentionally) confusing is to reuse the same identifier names in different scopes. The reviewers suggested that Pthin could forbid shadowing of identifiers, and perhaps even forbid using `self.foo` and `foo` in the same context. For example, **Navigator.execute()** uses both `self.selections` and `selections`, which some reviewers found tricky to follow.

One reviewer suggested the principle of never reusing a variable name for two different purposes. For example, **Navigator.play()** uses the local variable `option_i` for different purposes on lines 98 and 104. This particular violation could be found by a static analysis that requires all loop counters to be unbound before the loop begins.

A possible language feature that would reduce this problem would be a requirement that the first binding of any variable be preceded with a keyword (such as `var` as in JavaScript). This

would force programmers to declare whether they expect each variable to be already bound or not.

Subsetting. The reviewers noted that it is useful for a programming language to provide easy ways to enforce that a given portion of a program is in a particular subset of the language. Examples of this are the extensible auditing features in E and Joe-E. If reviewers can rely on static checkers to ensure that parts of a program are in declared subsets of the language, that can make their job as reviewers much easier.

Type distinctions. Python has no truly separate Boolean type; Boolean values behave in almost all respects like the integers 0 and 1. The reviewers suggested that it might be good for Pthin to treat integers and Boolean values as separate types and statically check that they are used in a type-safe way. There are a few places in the current Pvote code that would violate such a type restriction, such as Navigator.py line 27.

One reviewer noted difficulty in telling whether a variable name such as `group_i` stood for an nullable or non-nullable integer. This could be addressed by a type distinction or a naming convention. One suggested naming convention uses the prefix `opt` for optional (i.e., nullable) variables.

Mutability. The reviewers suggested that it would be useful to be able to declare some variables “eventually read-only.” Such variables would be initially mutable, but at some later point irreversibly become immutable (either upon exiting a particular scope or upon being marked immutable by the Pthin program). These could be used to ensure that the ballot definition is read-only after it is loaded and verified. An alternative would be to construct the ballot definition only out of immutable objects.

Another potentially useful behaviour that the reviewers suggested was a variant on Java’s `final` keyword: a variable that, after initialization, can only be set to `None`. Thus, it would be possible to “throw away” the variable as a way of divesting authority, but not to change it.

The reviewers also suggested that Pthin might require constructors to set all the member variables of the object being constructed.

Compilation. The reviewers suggested that instead of verifying the compiler, auditors could verify that the assembly-language output from the compiler is a valid compilation of the source code input to the compiler.

If Pthin were small enough, perhaps it could be reliably mechanically translated to a variety of target languages.

Other languages. The following other programming languages were suggested for implementing Pvote:

- BitC
- CCured
- Cyclone
- Java
- Joe-E (subset of Java)
- Ada
- SPARK Ada (subset of Ada)
- ML

In addition, JML (Java Modelling Language) declarations could be added to an implementation in Java or Joe-E, and verified by a static checker such as ESC/Java2.

Porting Pvote to Joe-E would help reviewers reason about statelessness and determinism (e.g., statelessness of the **Ballot** constructor or determinism of the verifier).

There is a trade-off here between choosing a well-known language (with a large community of potential code reviewers) and a more obscure language with verification features. The importance of public confidence in the election affects this trade-off.

Other language features. The reviewers mentioned that static typing and explicit control over memory allocation could be potentially helpful language features for the design and review of Pvote.

The reviewers wondered if it might be possible to further reduce Pthin by eliminating negative integers and strings, thereby making it easier to translate into other languages.

Also, there are a few places where Pthin had to be a slightly larger language in order to accommodate an existing API. An alternative to this would be to create an abstraction around the API, implement the abstraction in Python, and use a call to the Python function in the Pthin program. (This example illustrates the benefits of flexibility in choosing language subsets.)

Memory usage. Section 7.11 of the assurance document attempts to provide an argument that the memory usage of Pvote is bounded. How would an actual upper bound on memory usage be calculated given a particular ballot definition? How might Pvote’s design and Pthin’s specification be changed in order to make such a calculation straightforward?

Hardware. For a voting machine that emits audio via a typical headphone port, there is a risk that the audio may be recorded in violation of voter privacy. In particular, if audio is enabled by default and most voters don’t use audio, a cable running from the audio port to a recording device may go unnoticed [61].

Accessibility. The only user input events Pvote understands are screen touches and button presses, not including their duration, movement, velocity, pressure, or release. In particular, Pvote cannot distinguish long and short presses or detect double-clicks. We need to identify the norms for input devices in the accessibility community; if timed features like this are needed, Pvote may have to be altered to support them. (One reviewer pointed out that some support for such features could also be provided by hardware, such as hardware that translates a long button press into one keycode and a short button press into a different keycode.)

One-button or other low-bandwidth input interfaces could require Pvote to be more aware of timing. One example would be an interface where “pause” is an input event; another would

be an interface where options are read off slowly one at a time, and the user signals when he hears the desired option. For these designs, we would want to be able to specify a separate timeout length for each state, and potentially also an arbitrary action (not just a transition) to be triggered on a timeout.

Use of pointers. The reviewers debated whether it would be an improvement to have the verifier, as it goes through the ballot definition checking array indices, replace the integer array indices with pointers to the referenced array elements. This would make it easier to be sure that the preconditions checked in the verifier match the preconditions on which the rest of Pvote relies. However, there is a good rationale for using indices instead of pointers, since passing indices transfers no authority. For example, other modules can pass indices into the printer module that will be used as indices into the text data, even though these other modules don't have access to the text data.

One reviewer suggested that rights amplification might be a possible solution (bringing together an opaque array object and an opaque index object would yield an array element). It might be tricky to make this work for parallel arrays, which Pvote uses.

Output. The reviewers discussed the possibility of declaring the output module to be a replaceable component, separate from Pvote. Thus the interface to Pvote would be: take a ballot definition file as input, produce a cast vote record as output. The output module would print or record the cast vote in whatever appropriate manner. There was no consensus on how the output interface should be defined.

Printing. The reviewers were concerned that the printing module is based around 7-bit ASCII, thus restricting candidate names to 7-bit ASCII. Alternatively, if the printing module were to print images instead of text, problems related to text encoding would go away. Several options were discussed:

- Print numeric identifiers instead of strings; the numbers would refer to the ballot definition. (But one useful purpose

of a printed record is to allow votes to be counted even if all electronic records are lost; this option lacks that feature.)

- Allow Unicode strings; pass them through opaquely to the printer. The printer module should export a validation method that checks whether strings are printable by the printer hardware (e.g., the printer might support only 7-bit ASCII, or it might provide a font that supports some subset of Unicode). This validation would be performed on all strings at ballot loading time to ensure they will be safely printable.
- Just print sprites; eliminate all strings from the ballot definition and from Pthin. Some possibilities:
 - For each sprite to be shown on the display, provide a corresponding black-and-white sprite for printing.
 - Restrict all displayed sprites to 1-bit black-and-white bitmaps, so the printer output can match it exactly. (This also has the fairness advantage that colour-blind voters will perceive exactly the same ballot as other voters.)
 - Allow both of the above approaches and add a flag to the ballot definition to let the ballot designer choose one of them.
 - Specify an algorithm for converting a colour image to a black-and-white image for printing. If the ballot designer chooses to use a colour sprite, it is their responsibility to make sure that its black-and-white conversion is legible.

System platform. The reviewers pondered what a minimal platform for Pvote would look like, and sketched out the following:

- Audio driver (hardware that plays from a memory-mapped buffer, with software that keeps the buffer full)
- Interrupts for all input devices (including touchscreen touches)
- Printer driver
- Storage driver (SD card, etc.)
- Single-threaded program

Code documentation. The Pvote code was presented to the reviewers without comments, for fear that comments might bias their evaluation. Some reviewers had opinions about this:

- Some reviewers felt that it would be nice to see comments in the code, and that leaving comments out of the code didn't make their job easier.
- One reviewer was glad that the comments were separated, because (a) more code fits on fewer pages, and (b) he was not being influenced by comments he could not trust. He felt that he was getting more benefit by being forced to reconstruct for himself the argument for why the code was correct.
- One reviewer would prefer to see the meaning of fields described in comments right in the code (like Javadoc).
- "Code that needs no documentation" is a myth; the code says *how*, but the comments say *why*.

A possible compromise would be to include comments in the code, and also offer a way for the reviewers to view the code with the comments hidden.

Tests. Adding a suite of unit tests and regression tests might help the reviewers perform testing, though it would constitute more code for them to audit.

Bug insertion

This section describes the bug insertion experiment that we conducted. On the third and fourth days of the review, the reviewers were given a new hardcopy of the source code containing bugs that David Wagner and I had inserted. We told the reviewers that we had inserted at least one bug in the code, and asked them to try to find it.

Since insider attacks are a major unaddressed threat in existing systems, we specifically wanted to experiment with this scenario. Therefore, we warned the reviewers to treat us as untrusted adversaries, and that we might not always tell the truth. However, since it was in everyone's interest to use our limited time efficiently, we settled on a time-saving convention. We promised to truthfully answer any question about a factual matter that the reviewers could conceivably verify mechanically or by checking an independent source—for example, questions about the Python language, about static properties of the code, about its runtime behaviour, and so on.

As we sought to craft bugs on the evening of March 30, David Wagner and I chose the following criteria to make the experiment more realistic:

- The bug had to conceivably enable an attack that would affect election results. We assumed that the attacker also had the ability to distribute a maliciously designed ballot definition.
- The bug had to conceivably escape detection in a live walkthrough test, such as a “Logic and Accuracy Test” for an election, which typically consists of going through the whole casting process for several ballots so that at least one vote is cast for each candidate.
- The bug could not violate the Python language definition.

We only considered bugs that individually met all these criteria.

David and I devised and inserted three bugs with varying levels of subtlety:

1. **Easy:** Lines 83-84 in `Navigator.py` are as follows.

```
83 if step.op == OP_REMOVE and selected:
84     selections.remove(option_i)
```

We removed `and selected` from line 84. The consequence is that an attempt to deselect an option using `OP_REMOVE` will crash if the option is not already selected. A ballot definition could use this bug to selectively crash the machine in a particular situation (e.g., to disenfranchise those who vote for a particular party). The ballot definition could still pass a walkthrough test and avoid crashing under normal circumstances by using a condition to prevent `OP_REMOVE` from being executed when the option is not selected.

2. **Medium:** Lines 78-79 in `Navigator.py` are as follows.

```
78 selections = self.selections[group_i]
79 selected = option_i in selections
```

We changed `selections` to `self.selections` in the second line (line 79). The consequence is that `selected` will always be 0, because `self.selections` is a list of lists, not a list of integers. The consequence is that `OP_ADD` will keep adding a selection to the list even after it has already been selected. So, in a contest with a `max_sels` of 3, for example, a voter could cast three votes for the same candidate. (Note that this bug could be caught by a static type checker.)

3. **Hard:** Lines 42-43 in `Navigator.py` are as follows.

```
42 if option.writein_group_i != None:
43     self.review(option.writein_group_i, slot_i + 1, None)
```

This is the recursive call within the `review()` method. The recursion only goes one level deep: the outer call displays the selected options within a contest, and the inner call displays the selected characters within a write-in. Thus, the outer call passes the write-in group to the inner call. We changed `None` to `cursor_sprite_i` in the recursive call on line 43. This takes the `cursor_sprite_i` index that was passed in (which would be a sprite the size of an option)

and passes it on to the inner call (which would attempt to paste it into a slot the size of a character). The ballot definition could set up a situation in which this size mismatch caused a sprite to exceed the bounds of the screen, causing the program to crash.

We decided to insert all of these bugs in a 100-line region of a single file, lines 11 to 109 of `Navigator.py`, and told the reviewers to look in this region. We did this both because the navigator was the most interesting in terms of the program logic and because we knew the reviewers would have limited time. The new version of the code that we gave the reviewers contained all three bugs, but we did not tell the reviewers how many bugs there were.

March 31. Three reviewers were present on March 31: Tadayoshi Kohno, Mark Miller, and Dan Sandler. Dan was already very familiar with Python; he worked separately. He found the “medium” bug about 35 minutes after he started his search, purely by manual inspection, saying the line “looked suspicious.” He then found the “easy” bug about 35 minutes later (70 minutes after starting). He hypothesized that the condition was incomplete by reading the code, then tested his hypothesis by running `Pvote` and finding a way to make the program crash.

The other two reviewers, Mark and Yoshi, worked together. They were less familiar with Python; one had spent the preceding two days learning about `Pvote`'s design and inspecting the code, and the other was encountering `Pvote` for the first time with the bugs embedded. About four hours into the review (not including a lunch break), they expressed some concern about the code near the “easy” bug. About ten minutes later, they noticed that the annotations to the left of line 83 didn't match the code. Another ten minutes later, they declared that they had found a bug (the “easy” bug). Part of what had caused them to inspect this region of code carefully was an attempt to systematically verify, one by one, each of the

assurance arguments given in Chapter 7 of the assurance document. They did not find the “medium” bug.

By the time the reviewers quit late in the day, none had found the “hard” bug, although there had been some questions about ways that cursor sprites could be used to conduct an attack. They had spent a total of about 20 reviewer-hours examining the version of the code with the three inserted bugs.

May 20. Two reviewers were present on May 20: Ian Goldberg and Tadayoshi Kohno. Ian found the “easy” bug about 130 minutes after starting his search, despite being new to Pvote. About 90 minutes later, after no more bugs were found, we decided to switch strategies. To test out the “read-write review” idea that Dan Sandler had previously proposed (see Section E), both reviewers would try to insert bugs into the code, and we would see if this helped them find the bugs that David and I had inserted earlier.

Yoshi spent the next 50 minutes inserting bugs into the code. I examined his altered code and, by manual inspection alone, was able to find the three bugs he inserted in about 30 minutes. (Of course, as the author of the code, I was uniquely familiar with it, so this doesn’t reveal much about the subtlety of the inserted bugs.) No more bugs were discovered for the rest of the day. By the end of the day, the reviewers had inspected the code for about 13 reviewer-hours.

Review process

This section describes ideas and suggestions regarding the software review process that came up during the review.

Viewing code. One reviewer remarked that he was much more effective at comprehending someone else's code when all the code was spread out on the wall in front of him, on paper. He found this surprising because he had spent the last 20 years editing code on computer screens.

Analysis tools. The reviewers mentioned that these tools would have been helpful to them:

- a static checker to verify that Pvote is written in the Pthin subset
- a checker for suspiciously similar (possibly mistyped) identifiers
- an information flow analyzer
- a static analyzer to determine the maximum possible call depth

Trust in the adversary. The reviewers mentioned on several occasions that it was difficult to maintain the requisite level of distrust in the programmer, especially when the programmer was present in the room and was a friendly face. The significance of the social relationship between programmer and reviewer is an important difference between code review for accidental mistakes and code review for intentional malice. The reviewers agreed that in an adversarial review, programmers should not socialize with the reviewers; perhaps they should even not be physically in the same room, or communicate only over a text-based communication channel. The reviewers believed that measures like these—to “dehumanize the enemy”—would help them maintain the necessary distrust of the programmer.

One reviewer noted that, although his suspicions were

raised during the bug-finding test by a missing annotation, he would have been easily tricked by a bogus annotation. He would not have bothered to check that the annotation was correct, since it appeared that the programmer had thought about the issue and claimed to offer some justification, and since every other time he had checked out an annotation, it did turn out to be valid. This weakness resulted from a combination of the tediousness of checking annotations and insufficient distrust in the programmer.

Reviewer fatigue. The reviewers generally felt that the point where one becomes tired of inspecting code comes long before one has subjected it to enough scrutiny. It might be a good idea to limit the amount of time spent per reviewer: the more familiar one becomes with it, the more confident and comfortable one becomes at making assumptions of correctness. One reviewer suggested that, since reviewers shouldn't ever become complacent with the code being reviewed, the review process should follow a "principle of most surprise" to keep reviewers on their toes.

One-line change test. Mark Miller proposed the following test: suppose that, as an attacker, you had the ability to change just one line of code. How much damage could you do (i.e., which assurance requirements could you cause the program to violate)? Figuring out which lines are the most sensitive would provide a map of the "hot spots" in the program—the places that require especially close attention during a code review. For example, changing `- 1` to `+ 1` on line 12 of `Navigator.py` is sufficient to make Pvote keep printing out ballots repeatedly if left unattended. Therefore, this line is part of the TCB for R3 (become inert after a ballot is committed) and also for R9 (commit the ballot only when so requested by the voter).

In a variant of this test, there are a series of trials. For each trial, one line of the program is chosen at random and the attacker is allowed to change just that line. With enough trials, one could estimate the size of the TCB for each assurance

requirement. For example, if the attacker is able to violate a particular requirement in 1/4 of the trials, then the TCB for that requirement is probably about 1/4 the size of the program.

By changing almost any single line, one can trivially cause the program to crash. It is more of a challenge to cause a meaningful effect on an election without failing a simple operational test.

Our discussion of the one-line change test highlighted the benefits of read-only types. Without read-only restrictions, almost any line in Pvote can be changed to one that maliciously modifies the ballot data in memory.

The read-write review. Dan Sandler suggested the possibility of taking the bug insertion experiment one step further by encouraging the reviewers to insert their own bugs, a process he called the “read-write review.” He conjectured that being tasked to insert bugs would:

- Motivate reviewers to find “hot spots” in the code that were especially vulnerable to small changes, thereby leading them to scrutinize places where malicious bugs were likely to have been inserted.
- Force reviewers to modify and run the program with the intention of producing a specific change in behaviour, thus requiring them to develop a deeper understanding of how the program works than they would get from merely reading the code.
- Yield a program with known bugs that could then be passed on to another group of reviewers to inspect. The existence of the known bugs would motivate the next group, and the fraction of those bugs they found could offer some measure of their effectiveness.

One could imagine several groups of reviewers performing a multi-round review, in which each group inserts some bugs and then passes on the code to the next group.

Other tasks might also improve code understanding by getting reviewers to modify and interact with the code. Reviewers could be asked to translate it to another

programming language, or to rewrite parts of the code they find hard to understand, and then verify that their rewritten or translated code produces equivalent behaviour.

The idea of the read-write review was inspired by Dan's experience with the Hack-a-Vote class exercise, in which more bugs were found by students while inserting bugs than while looking for bugs. The insight was that although Hack-a-Vote was conceived as a test of the students doing the hacking, it is also a test of the Hack-a-Vote software's resistance to undetected tampering.

Ideally, if reviewers find most or all of the planted bugs, while finding few or no bugs in the original code, this might be grounds for confidence in the original code. However, we noted several ways that an actual attacker (the original, possibly malicious programmer who initially wrote the software) might be a stronger adversary than a fake attacker (a code reviewer asked to insert bugs into the software):

- A real attacker could simply be smarter.
- A real attacker may be more motivated or have more at stake.
- A real attacker may have more time and resources than a team of reviewers would have in one round of the review.
- A real attacker would be more familiar with the code, and could have chosen the design and implementation specifically to enable particular malicious bugs.

On the fourth day of the review, reviewers were asked to insert their own bugs. They commented:

- It's possible that inserting bugs may reduce a reviewer's chances of finding bugs. Inserting bugs under time constraints may encourage reviewers to stick to the parts of code they already understand well, instead of diving deep into unfamiliar parts of the code.
- The code can be divided into three classes: (a) parts you understand, (b) parts you don't understand, and (c) parts you don't understand but think you do. Reviewers will tend to insert bugs in types (a) and (c), but not (b).

Post-review survey

After the conclusion of the first three-day meeting, we informally surveyed the reviewers by e-mail. Their responses are paraphrased here.

Thoroughness of review . *How thorough was this review, compared to other security reviews you have participated in, or other reviews of voting software?*

- This was comparable to other code reviews, though very different from reviewing commercial voting software because Pvote is so much smaller.
- Other reviews expended more total effort, but this review spent more effort per line of code.
- This did not go into as much depth as other security reviews because we were focused on just the Pvote component.
- For me, not that thorough.

General confidence. *After this review, how much confidence do you have have in Pvote, compared to other voting systems you are familiar with?*

- Much more confidence in Pvote than any commercial voting system; however, Pvote is only one component and many of the security flaws in other voting systems occur in parts outside of Pvote’s scope. “Comparing Pvote to the comparable portions of commercial systems is no contest. Pvote kills them.”
- For what Pvote does, much better than any of the other systems I have seen.
- I’m not familiar with other voting systems.
- I can’t give a confidence level about Pvote, though I am confident it would be easier to argue the security of Pvote than other designs.

Lack of accidental bugs. *How confident are you that Pvote is free of accidental bugs? In other words, if you assume that Ping*

is not malicious and was trying his best to make Pvote trustworthy, how confident are you that you would have found any inadvertent bugs in Pvote?

- Reasonably confident.
- Rather highly.
- Confident due to the efforts of the group as a whole, though not very confident I would have found them on my own.
- It's hard to say.

Lack of malicious bugs. *How confident are you that Pvote is free of malicious code? In other words, if you assume that Ping is malicious and may have been trying his best to introduce a backdoor, how confident are you that you would have found it?*

- Not at all confident.
- Poorly.
- Confident due to the efforts of the group as a whole, though not very confident I would have found them on my own.
- Not very confident.

GNU Free Documentation License

1.2, November 2002

Copyright © 2000, 2001, 2002 Free Software Foundation, Inc.
51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. Applicability and definitions

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “**Document**”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “**you**”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “**Modified Version**” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “**Secondary Section**” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “**Invariant Sections**” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The “**Cover Texts**” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A “**Transparent**” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not “Transparent” is called “**Opaque**”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “**Title Page**” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section “**Entitled XYZ**” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “**Acknowledgements**”, “**Dedications**”, “**Endorsements**”, or “**History**”). To “**Preserve the Title**” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. Verbatim copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. Copying in quantity

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until

at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- D. Preserve all the copyright notices of the Document.
- E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- H. Include an unaltered copy of this License.
- I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
- N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
- O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made

by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. Combining documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. Collections of documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. Aggregation with independent works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. Future revisions of this license

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

Addendum: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the “with . . . Texts.” line with this:

with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.