Prerendered User Interfaces for Higher-Assurance Electronic Voting

Ka-Ping Yee[*] ping@zesty.ca
David Wagner[*] daw@cs.berkeley.edu
Marti Hearst[*] hearst@sims.berkeley.edu
Steven M. Bellovin[†] smb@cs.columbia.edu

Abstract

We propose an electronic voting machine architecture in which the voting user interface is prerendered and published before election day. The prerendered user interface is a verifiable artifact — an *electronic sample ballot* — enabling public participation in the review, verification, usability testing, and accessibility testing of the ballot. Preparing the user interface outside of the voting machine dramatically reduces the amount and difficulty of software verification required to assure the correctness of the election result. We present a design for a high-assurance touchscreen voting machine that supports a wide range of user interface styles and demonstrate its feasibility by implementing it in less than 300 lines of Python code.

1 Introduction

Democratic elections are increasingly depending upon electronic voting systems. In 2002, the United States passed the Help America Vote Act [1], which includes a requirement for "at least one direct recording electronic voting system or other voting system equipped for individuals with disabilities at each polling place." Over $300 million in federal funds has been disbursed specifically to pay for new voting machines [15]. Many other governments around the world are planning ahead for large-scale deployment of electronic voting.

Electronic voting machines have the potential to provide significant improvements in usability and accessibility over paper ballots. For example, they can be designed to help voters detect and correct mistakes; they can provide alternate user interfaces for individuals with disabilities; and they can be programmed with support for more language choices than a typical paper ballot. However, the electronic voting process lacks the transparency of paper voting, the correct functioning of a computer program is difficult to assure, and computer failures are an everyday part of modern life. Moreover, elections are an especially high-profile and potentially rewarding target for attack, and a broad range of parties stand to benefit from influencing their outcome.

The typical challenge in software security is to design software to defend against various threats. However, because the stakes are so high for electronic voting, the threat model must include the possibility of malicious code in the voting system. Even in the absence of deliberate insider fraud, well-intentioned programmers can make mistakes. Thus, our challenge is not only to design a secure voting machine program, but also to design an overall architecture for the election system that lets us confirm that it really is secure.

Though software is involved at many stages of the election process, this work focuses on the software in the voting machine itself. We will explain in Section 3.2 why we believe this to be the most critical software component of the system. Unfortunately, the software in today's voting machines is far too large to allow automated verification or thorough independent review, given the time and cost constraints of the election equipment certification process. In 2004, Kohno, Stubblefield, Rubin, and Wallach [5] examined the source code for the Diebold AccuVote TS machine and found it to contain many serious design and engineering errors, declaring it "far below even the most minimal security standards applicable in other contexts." The main AccuVote TS program consists of over 31,000 lines of C++ code and resource scripts, ignoring comments and blank lines. Verifying the correctness of a program this size is overwhelmingly difficult.

We observe that the user interface (UI) is a major contributor to software complexity. By our estimate, the voting UI constitutes about 14,000 lines of the aforementioned source code. The key idea we propose is to construct and verify a prerendered description of

[*]University of California, Berkeley, CA, 94720
[†]Columbia University, New York, NY 10027

the UI before the election. Prerendering the UI yields several significant advantages:

- It simplifies the software running in the voting machine, facilitating its verification.
- It mitigates the conflict between accessibility and security concerns by enabling the UI design to be highly flexible without affecting the security properties of the machine.
- It mitigates the conflict between the proprietary interests of voting machine vendors and the public benefits of transparency by reducing the portion of code that has to be disclosed to evaluate the security of the machine.
- It enables the UI to be updated and verified independently of, and more easily than, the voting machine software.
- It allows the UI to be separately published and to be run on commodity hardware, enabling it to be tested by anyone — not just those with access to the equipment that will actually be used on election day.

In this paper, we propose an electronic voting machine architecture based on the concept of a prerendered user interface. We present a specific software design for a touchscreen voting machine in this architecture, describe our prototype implementation of the machine, and evaluate this implementation in terms of its security and verifiability.

2 Goals

We begin by identifying six high-level goals for a secure, verifiable, and usable election architecture.

1. **Minimize trusted code.** Reducing the amount of trusted code (the portion of code that needs to be verified) makes software verification easier and more reliable. By "verification" we mean informal code review, independent security audits, formal methods, and everything in between. All these kinds of software verification are highly sensitive to code size, because small changes can have far-reaching effects and software components can interact in unexpected ways.
2. **Design for verification.** The difficulty of software verification is reduced by designing code and data structures specifically to make them more amenable to analysis. Examples include componentization and limited data flows.
3. **Minimize code churn.** If the trusted code changes infrequently, each release can be tested and audited more thoroughly. Hence, voting systems should be designed so that customization or new functionality can be provided without changing the trusted code.
4. **Support public review.** The success of a democratic election depends not only upon the actual reliability of the voting system but also upon public confidence in that reliability. Therefore, the election system should allow as much as possible of the election to be verifiable by the public, including non-programmers.
5. **Support accessibility.** The architecture should allow for user interfaces that enable individuals with disabilities to vote privately and independently and should facilitate their participation in reviewing and testing these user interfaces.
6. **Support interoperability.** Election officials should be able to mix and match components from many vendors. To this end, the system should define clear interfaces between components. This enhances the effectiveness of testing, as components can be tested in isolation and multiple implementations of a component can be checked against each other. Also, the resulting market competition may reduce election costs.

The following are some basic requirements of democratic elections:

- Each voter may only vote once, and only in contests for which the voter is authorized.
- Votes must be reported accurately.
- Each voter's choices must be kept secret.
- The voting system must not provide opportunities for voters to sell their votes or to be coerced into voting a particular way.
- The voting system must work reliably.

Standard ballot features that the voting system should support include "vote for k out of n," write-ins, multiple languages, and straight-ticket voting, and the system should not preclude the possibility of ranked voting. Electronic voting systems are also expected to prevent voters from casting an *overvote* (choosing too many selections) and to notify voters if they are about to *undervote* (choose fewer than the allowed number of selections).

In this work, we focus on producing voting machine software that works correctly and verifying that it works correctly. Other parts of the election system such as absentee ballots, voting by mail, and voter registration are outside of our scope. To run an accurate election, it is also necessary to make sure the machines are actually running the software that was approved and to protect the voting machine and its storage media from tampering. We do not address physical security and chain-of-custody issues in this paper.

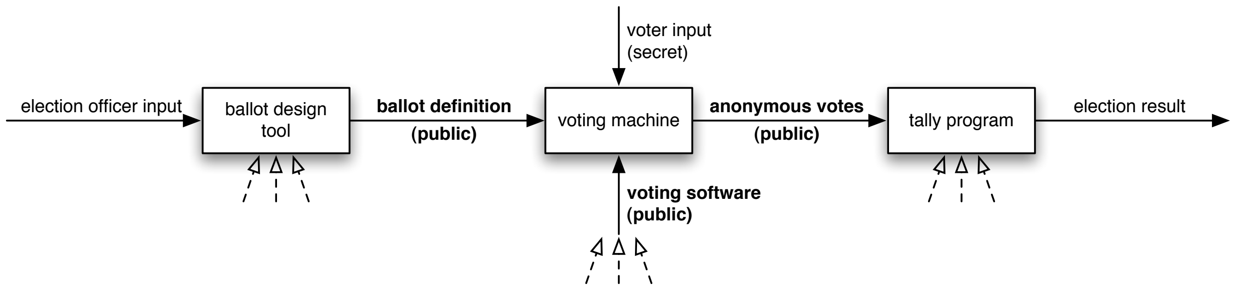


Figure 1: Simplified block diagram of the proposed election system architecture. The dashed arrows hint at the complex web of dependencies (including source code, operating systems, compilers, editors, and other tools) underlying each software component. Publishing the inputs and outputs of the DRE (shown in bold) lets us cut away these dependencies when performing a security evaluation.

3 Architecture

Election systems depend on software for many different functions before, during, and after the actual day of the election. Because so little information is typically published about the programs used to conduct an election and their inputs and outputs, trusting the outcome of a computerized election today requires trusting nearly everything in the system — including the software that produces ballot definitions, the voting machine software, the software that tallies votes, and all the operating systems, compilers, editors, and other tools that were used to produce these programs. In the following sections, we present a general election system architecture that reduces what must be accepted on faith to trust the validity of the election result.

3.1 Verification Methods

If a software component has input x and output y, and if it is supposed to implement a deterministic function f, there are two ways to check that the component has produced the correct output:

1. **Program verification.** Examine the implementation of the software component and verify that it matches the specification of f. This may include manual source code analysis, formal verification, or other methods.
2. **Result verification.** Given x, compute $f(x)$ and check that it matches the actual output, y. Doing this requires records of both x and y and an independent implementation of f.

Program verification only needs to be performed once for a given implementation of f, whereas result verification must be performed for each time f is executed. However, proving statements about the behaviour of software programs is generally very difficult. The state of the art in automated software verification can only verify small programs of limited complexity, against specifications that are difficult to write. Software review by human experts is time-consuming and can be prone to error. Furthermore, program verification requires disclosure of the software code, which often faces legal, financial, or political barriers. Disclosing code always improves the transparency of the process, but it is useful to be able to check the correctness of an election without *requiring* inspection of all the code. For all of these reasons, our architecture is designed to minimize dependence on program verification.

3.2 Election Verification

“Direct recording electronic” (DRE) is the industry term for the machine that handles voter input and recording of votes. The DRE is the first step in the chain; its input is a human interaction. Election rules forbid “leaking” vote information that can be identified with specific voters, so the human interaction must be kept secret. Consequently, result verification of the DRE is not an option; program verification is the only way to gain trust in the DRE.

However, if the DRE stores votes in anonymous form, its output can be published. In addition, the inputs and outputs of all other components of the system can be published, and so can be checked by result verification. Thus the only part of the process that requires program verification is that part ranging from the input of the voter’s selections to the point where the selections are recorded anonymously.

Therefore, our approach is to minimize the size and complexity of the DRE software (even if it means that other components become more complex) and to publish all of the DRE’s inputs and outputs — except for the votes themselves, until they are anonymized — to enable result verification of the rest of the process (Figure 1).

The ballot definition is published far enough in advance that it can be validated before election day. For instance, the ballot definition might be published on government websites and made available to candidates; anyone would be able to download it and run 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. In this way, the published ballot definition is analogous to the paper sample ballot typically mailed to voters before an election.

The anonymized cast vote records from every DRE are published for all to see after the election. Anyone can add up the votes in these files to obtain the election-wide totals and compare them against the official totals to gain confidence that tallying was done correctly. Also, pollworkers and observers might be encouraged to check the summary tapes that are printed at the close of polls against the published electronic vote files to verify that the files were not tampered with while in transit.

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

3.3 Prerendering the User Interface

In a typical DRE, much of the software code is responsible for generating the voting user interface in real-time on the running machine. This includes the code for arranging the layout of elements on the screen, rendering text in a variety of typefaces and languages, drawing buttons, boxes, icons, and so on.

The DRE software can be considerably simplified by moving this layout and rendering functionality into a separate pre-election component. Instead of a ballot definition (such as those used by today's DREs) that lists only essential information about contests and candidates, we propose a ballot definition that describes the entire user interface. For a visual interface, this would include prerendered images of the screen and interface elements exactly as the user will see them; for an audio interface, this would include prerecorded sound clips.

There is some precedent for using prerendered bitmaps in electronic voting machines. For example, the ES&S iVotronic uses bitmap ballots [3], which help provide flexible support for different languages. The ballot definitions we propose contain not just the prerendered images but a complete description of the user interface — the locations where the images will appear, the transitions from screen to screen, how these transitions are triggered, and so on.

3.4 Virtual Machine

In our architecture, the ballot definition is a high-level, platform-independent description of the user interface for voting, displayed by a *virtual machine* (VM) that provides a high-level interface to the input and output hardware. 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 DRE hardware platforms would enable all of them to interoperate using the same formats for ballot definitions and recorded votes. We hypothesize that the VM implementation can be made considerably smaller, simpler, and easier to verify than the software in today's DREs.

Our proposal can be compared to the previously proposed "frog" voting architecture [2]: both are motivated by a similar desire to reduce the size and complexity of the trusted base on which the security of the voting system rests. The frog architecture separates the voting process into two steps: vote generation and vote casting. The voter first selects their votes on the vote-generation machine, which stores them on a "frog" (a storage device). Then the voter puts the frog into the vote-casting machine, which displays the contents of the frog for the voter to check, and upon confirmation, casts the votes.

A key assumption of the frog architecture is that responsibility for security rests on the simpler vote-casting machine; the vote-generation machine will have "no need for high security" [2]. This assumption requires that we rely on voters to check their frogs carefully before casting them. But some voters may give the vote-casting machine only a cursory glance, and most are likely to be influenced by confirmation bias [9], so it remains possible that votes recorded incorrectly by the vote-generation machine would go unnoticed. Even if voters are willing to check their votes carefully, the vote-generation machine remains in a position to influence voters during the selection process. For example, the vote-generation machine could present the options 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 give misleading instructions to voters, such as telling them to ignore the vote-casting machine or to go to a different polling place.

Our proposed architecture therefore targets a broader security goal: we wish 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 UI is not incompatible with a further partitioning of the user interface into two steps as suggested by the frog voting architecture.

3.5 Electronic Sample Ballot

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 consumer PCs 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.

As we mentioned in the preceding section, 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 influence the voter input. The published electronic sample ballot gives the election a *verifiable user interface*, which can be examined and tested by all voters, members of the disabled community, usability experts, and accessibility experts.

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 recent, 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 [4], which made it impossible to cast the Chinese ballot. A published sample ballot 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.

3.6 Ballot Definition Visualization

Running the ballot definition in a live test might show that the ballot appears to behave correctly, but it would not be a sure way to test the complete behaviour of the ballot. 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.

Therefore, we propose a software tool that transforms 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.

3.7 Anonymous Recording

We return now to two security requirements mentioned previously: voter privacy and coercion prevention.

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.

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, 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. Splitting up the ballot would conflict with election rules in some states that require the entire ballot to be recorded intact; on the other hand, it could be argued that a constitutional right to a secret ballot takes priority over state regulations.

4 Design

This section describes our current design for a touchscreen voting machine based on the above architecture that comes close to the richness and capability of today's touchscreen voting interfaces for sighted voters. This design supports only a visual interface, but could be extended to support audio or braille interfaces for visually impaired, blind, or deafblind voters.

A traditional method of recording the voter's selections is to store a numeric code or a text string identifying each selected candidate. Instead, we store the image containing the candidate's name exactly as it was shown to the voter, or for a write-in, the sequence of images of the characters selected by the voter, to reduce the risk of confusion.

Our design allows the voter to choose one or more options from a list of options, which is sufficient to emulate any choice that could be expressed by selecting bubbles or arrows on an optical-scan ballot. We discuss ways to support ranking of options in Section 7.5.

4.1 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 (Figure 2). 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 confirm the accuracy of a language translation) is easier than checking the correctness of a ballot model.

4.1.1 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 elements can be *activated* by the user. In a touchscreen interface, these elements correspond to rectangular areas of the screen that are activated by touches.

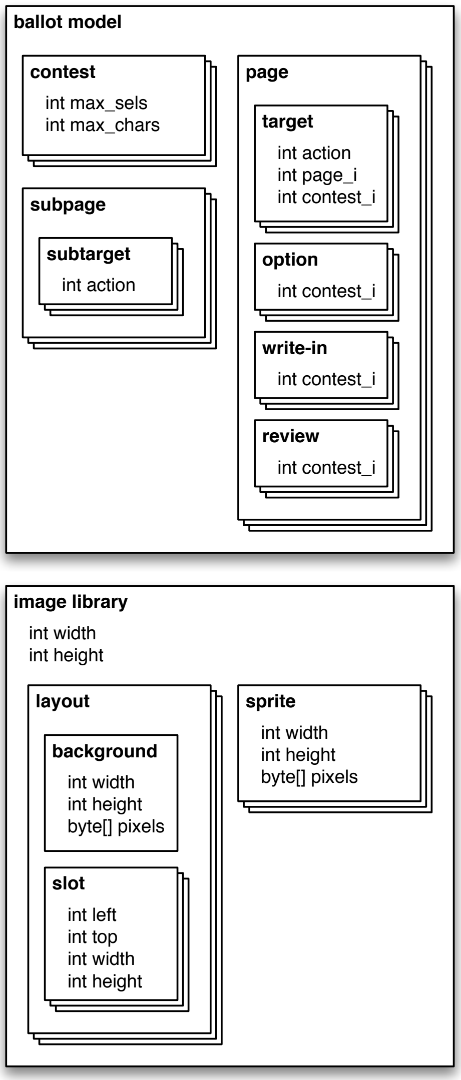
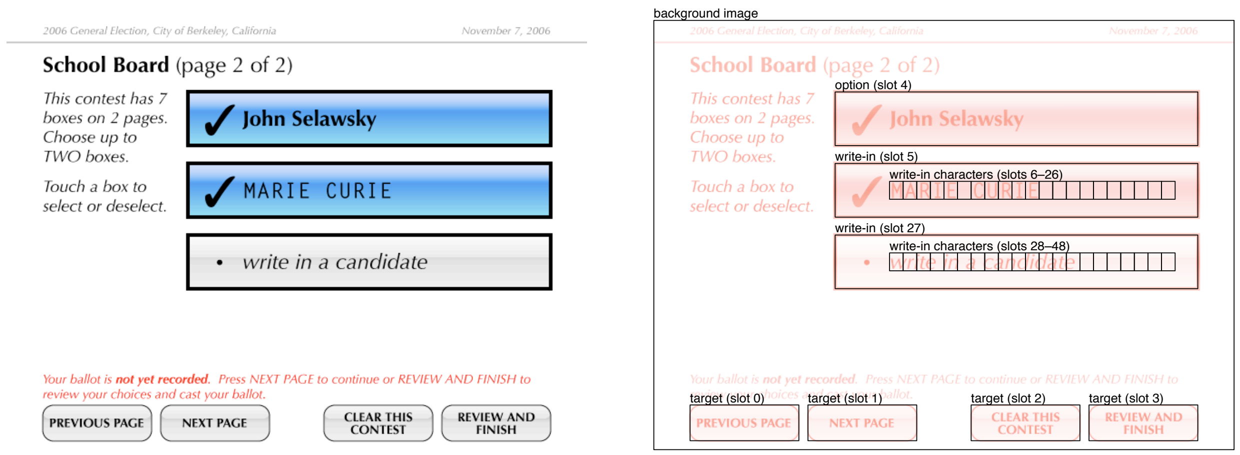


Figure 2: Structure of the ballot definition. Names ending with _i indicate array indices.

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



2006 General Election, City of Berkeley, California November 7, 2006

School Board (page 2 of 2)

This contest has 7 boxes on 2 pages. Choose up to TWO boxes.

Touch a box to select or deselect.

*Your ballot is **not yet recorded**. Press NEXT PAGE to continue or REVIEW AND FINISH to review your choices and cast your ballot.*

Figure 3: An example of a selection page with two options currently selected, and its corresponding layout.

it between a selected state and an 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 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.

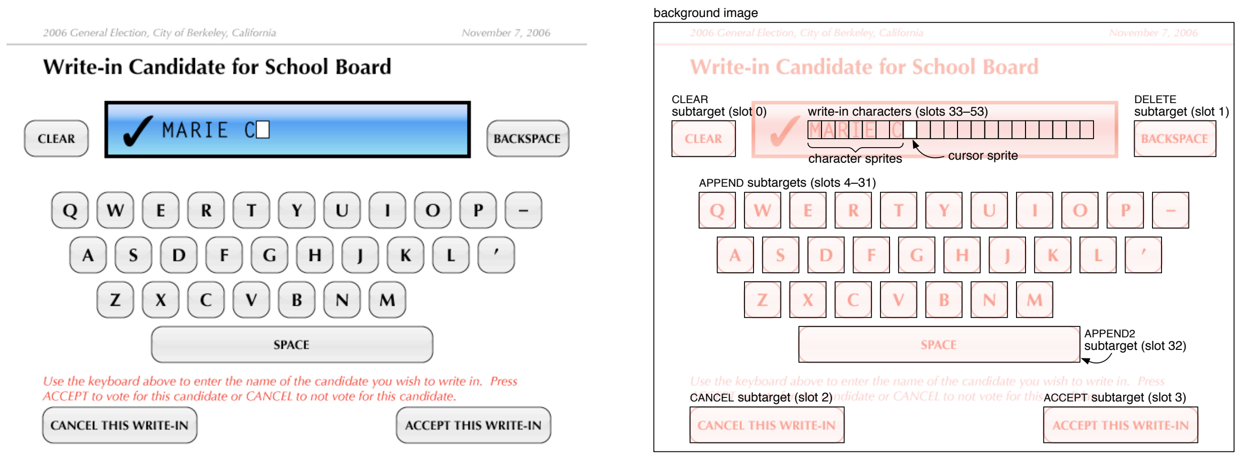
Because an ACCEPT subtarget only works when there is write-in text present, a write-in cannot be simultaneously empty and selected. The purpose of APPEND2 is to prevent a write-in from *appearing* empty and yet being selected. For example, if the keyboard's "space" button is an APPEND2 subtarget, then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

4.1.2 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 can be typed into a write-in, and the image of the text entry cursor shown while entering a write-in. To keep the DRE software as simple as possible, all images are stored uncompressed with 3 bytes per pixel.


2006 General Election, City of Berkeley, California November 7, 2006

Write-in Candidate for School Board

Use the keyboard above to enter the name of the candidate you wish to write in. Press ACCEPT to vote for this candidate or CANCEL to not vote for this candidate.

Figure 4: An example of a write-in subpage with a few characters entered, and its corresponding layout.

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_sels groups of slots (for displaying up to max_sels 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.

4.1.3 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 $1 +$ max_chars slots for each write-in, then max_sels $\times (1 +$ 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.

4.1.4 Well-formedness and Validity

We distinguish two different notions of the correctness of a ballot definition. A ballot definition is *well-formed* if it satisfies the assumptions made by the virtual machine implementation. A ballot definition is *valid* if it represents an acceptable user interface for voting.

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 contains 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 Section 4.1.3.
- 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 Section 4.1.3.

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 we expect to be common, as they prevent some obvious pitfalls and 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 contains 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.
- Before casting the ballot (arriving at the last page), the user must be shown pages that contain reviews for all the contests.

The ballot design tool could provide guidance, enforce validity conditions, or give notification when validity conditions are not met.

4.2 Virtual Machine

The VM is composed of four software modules: the *navigator*, the *video driver*, the *event loop*, and the *vote recorder* (Figure 5). 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. Rather, the separation into modules is an instance of design for verification. Establishing limited responsibilities for each module and limited data flows among modules facilitates the auditing and testing necessary to eliminate vulnerabilities to attack.

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.
- 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_sels 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:

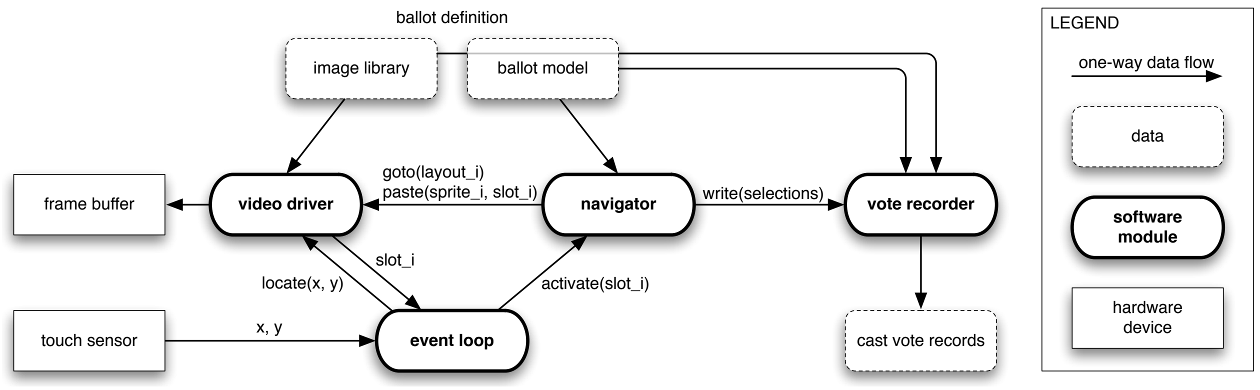


Figure 5: Block diagram of the virtual machine, which consists of the four software modules in bold. The arguments layout_i, sprite_i, slot_i, x, and y are integers; selections is an array of arrays of lists of integers.

- 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. (If the point lies within an overlapping target and review, the target slot will be returned because targets come first.)

The **event loop** receives touch events from the screen's touch sensor. We assume 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 [7] 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.

5 Implementation

To evaluate the feasibility and complexity of our voting machine design, we built a prototype implementation in Python [12] that runs on Linux, MacOS, or Windows. Our prototype uses Pygame [11], 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.

The prototype 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.

Our prototype 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. We assume 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 our prototype implementation and a sample ballot definition file are available online at `http://zesty.ca/voting/`.

5.1 Ballot Definition File

A separate Python module, not shown in Figure 5, 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 (red, green, and blue) for each pixel.

The prototype 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.

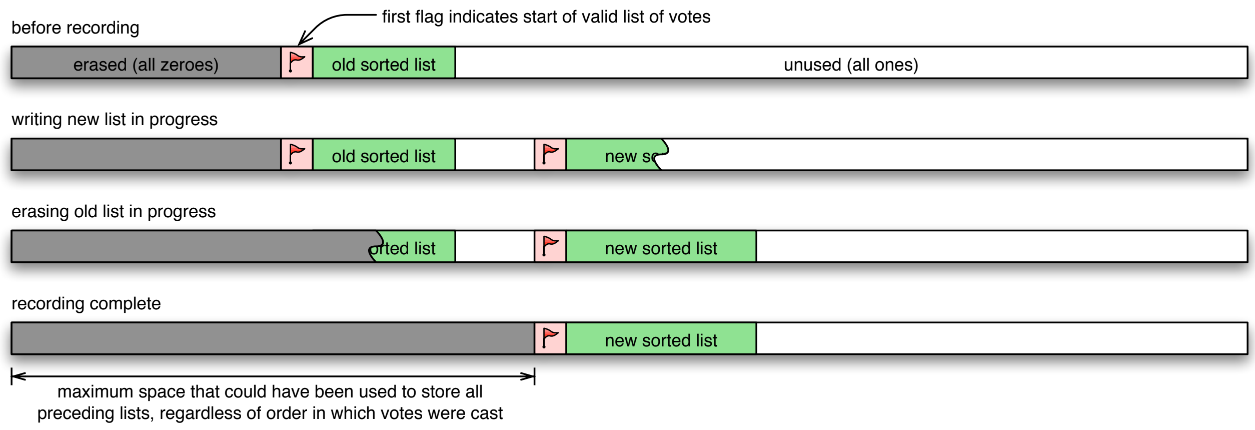

first flag indicates start of valid list of votes
before recording
erased (all zeroes)
old sorted list
unused (all ones)
writing new list in progress
old sorted list
new so
erasing old list in progress
orted list
new sorted list
recording complete
new sorted list
maximum space that could have been used to store all preceding lists, regardless of order in which votes were cast

Figure 6: 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.

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.

We leave open the question of how the pollworker's selection would be implemented in hardware. 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, our software prototype models this step simply as having the authorized choice of `ballot` file be present when the program starts.

5.2 Vote Storage File

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.

For this prototype, we have chosen to store the ballots using a *copyover list* [7], 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 we add items to the list, we write a new copy of the entire list in sorted order and erase the old copy by overwriting it with zeroes. Because the items are stored in sorted order, 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 [7] 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 [8] 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 3.7.

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 casting the long one first would yield a larger erased space than if they were cast in the opposite order.) Therefore, we 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 6. This commitment is atomic, because changing even one bit invalidates the flag.

5.3 Interpreting Recorded Votes

For a stored selection to have well-defined semantics, it must be somehow associated with a ballot definition. We considered four 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 background image for a page 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 several 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.

We chose the third scheme for our prototype because it is space-efficient, 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. However, in order to ascertain the true meaning of a vote, one must otherwise obtain a copy of the ballot definition. Our architecture assumes that the ballot definitions are published, so this is not a serious problem.

The fourth scheme stores the actual ballot definitions, yielding a vote record that is fully self-contained. But in order to store all the definitions on write-once storage, without revealing any information 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 an implementation where the machine 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.

6 Evaluation

6.1 Size

The entire prototype implementation is 293 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	94 lines
video driver	22 lines
subtotal (user interface)	255 lines
vote recorder	38 lines
total	293 lines

6.2 Dependencies

Our prototype runs on Python version 2.3. We have tried to minimize the dependencies in our implementation so that the size of the Python code gives a reasonable indication of the true complexity of the program. We use only one collection type, the Python list. Although some lists change length during runtime, 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.

6.2.1 User Interface Modules

The user interface modules import nothing from Python's standard library and use only the following 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 we use is **blit**, which copies a bitmap onto the screen.

6.2.2 Vote Recorder Module

The vote recorder 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.

6.3 Functionality

Our design allows a wide range of possible ballot formats. For instance, our prototype can support:

- 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 our 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.

Our design does not support an audio interface or a printed record; these are discussed in Section 7. It does not support straight-ticket voting, ranked voting, cross-endorsed candidates, automatic ballot rotation, or generation of audit logs, though it could be extended to include these features.

Our prototype 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.

6.4 Separation of Concerns

Our prototype 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 lessens our 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.

6.5 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, our prototype 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.

6.6 Comparison

At only 293 lines of Python, our prototype code is much smaller than the 31,000 lines of C++ in the AccuVote TS. It may be slightly more appropriate to compare our 255 lines of UI code with the AccuVote's 14,000 lines of UI code — but neither comparison is entirely fair, because our prototype lacks some of the AccuVote's functionality and the two systems have different sets of dependencies. Nonetheless, the correctness of our code 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 we have less code is our choice of programming language. Our prototype requires a Python interpreter, whereas the AccuVote TS does not. On the other hand, the 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 that the **blit** functionality we use 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 Python interpreter [10] that fits in a 504-kilobyte installation package, which also includes over 40 Python library modules that we do not use.

Alternatively, the Python code could be translated into a compiled language. Although we did use a higher-level language, we have been careful to minimize our use of Python's library modules and built-in functions, as described in Section 6.2. It is reasonable to expect that translating our code into a compiled language would multiply its size by a factor of 3 or 4, but not by 100.

Despite its small size, our prototype 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 our prototype, due to its design. The AccuVote TS code does not lend itself to similarly easy analysis.

7 Open Issues

This section sketches out some of our ideas for ways to add important missing functionality to our design. Our intention is to show that the basic architecture we have described does not pose fundamental obstacles to adding these essential features, not necessarily to present optimal solutions for achieving them.

7.1 Accessible Interfaces

One way to provide an audio interface would be to add a *sound library* to the ballot definition, containing prerecorded audio clips of spoken instructions, contest descriptions, and candidate names. A new module, the audio driver, would play clips from the sound library upon request by the navigator.

The event loop would handle user input from hardware buttons, and the ballot definition would specify additional targets for handling button presses. Extending the event loop to support hardware buttons would also be a way to support alternate input devices for voters with physical disabilities; the voting machine could provide a standard hardware interface for plugging in a wide variety of switch inputs.

Combined video and audio interfaces can be very helpful for users with impaired vision and we aim to provide synchronized video and audio in our future work on an accessible design.

Although the majority of blind individuals do not use braille, a braille interface would provide access to deafblind voters and improve access for those who prefer braille. This might be implemented with the addition of another ballot definition component containing data to be sent to a braille display.

7.2 Printing

Our design could be extended to produce a voter-verifiable ballot record by adding a print driver module that controls the printer. For a DRE with a prerendered user interface, the printout might contain either the exact images that the voter saw or printed text representing the voter's selections.

For a graphical printout, the print driver module would have access only to the image library, and would tell the printer to print the sprites that the user selected.

One way to support a text printout would be to add a *dictionary* component to the ballot definition that associates each contest and sprite with a string. Only the print driver module would have access to the dictionary; it would send these strings to the printer to describe the user's selections.

7.3 Audit Logging

The division of the software into modules makes communication among the modules a natural place to introduce audit logging. An audit log would record the ballot definition together with the sequence of all the messages sent between modules. The audit log would not normally be published, but in the event of a dispute, it could be used to replay the user interaction sequence to reveal both software errors and voter errors. Note that to protect voter privacy, the interaction sequence for each user must be protected in the same fashion as the actual ballots cast.

7.4 Straight-Ticket Voting

The basic concept of straight-ticket voting can be implemented by providing a target with an action that sets all the user's selections to a preconfigured state, though election rules may affect whether and how voters should be able to specify exceptions to these presets. The design of such a feature will depend on research into the various rules for straight-ticket voting in different jurisdictions.

7.5 Alternate 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 underrepresents centrists and often motivates voters to misrepresent their preferences [6], locking in polarized two-party control of the government.

One simple way to obtain a truer representation of voter preferences is approval voting, in which each voter can vote for as many candidates as they want. An approval election is easily conducted with our prototype by setting max_sels equal to the number of candidates.

Other improved election methods, such the Schulze method [13] or the Tideman method [14], use voters' rankings of the candidates to achieve a fairer result. Our current design does not directly support ranking of options, though ranking could be crudely implemented by repeating the same list of options, as in some paper elections. For example, San Francisco's ranked ballots show the same list of candidates 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. However, since our existing prototype knows nothing about the semantics of ranking, it cannot warn the voter about invalid rankings.

To provide proper ranking support, our design could be extended to include images of numbers in the image library and to display numbers next to ranked options. The navigator could initially assign successive rank numbers as the voter makes multiple selections; to reorder the rankings, the voter could clear the contest and start again or press special targets to increment and decrement ranks. Alternatively, a subpage with a numeric keypad could be provided for typing in the rank numbers.

7.6 Ballot Definition Tools

We have not yet built the design tool for producing the ballot definition or the visualization tool for verifying ballot definitions. The existence of the ballot definition as a separate artifact opens up possibilities for interesting new research in automated description, validation, and evaluation of user interfaces. The design tool is in a position not only to check for validity according to election regulations, but also to compute measures of usability and accessibility and provide guidance to the ballot designer during the layout process.

8 Conclusion

We have presented an electronic voting machine architecture capable of offering much stronger levels of assurance in both its software implementation and its user interface with considerably less verification and testing effort compared to an existing electronic voting system. Our architecture also provides broader public access to the verification process and has the potential to level the playing field for voters with disabilities and other minorities. In addition, we have presented a specific design for a touchscreen voting machine and have demonstrated that it can be implemented in a small fraction of the amount of code in current voting machines.

9 Acknowledgements

We thank David Dill for enlightening discussions about design for verification; the categories "program verification" and "result verification" were suggested by him. We thank David Jefferson for bringing the importance of interoperability to our attention and Dan Wallach for pointing out that bitmaps are already used by some machines. We are grateful to Scott Luebking for his helpful advice on several usability and accessibility considerations. Finally, we thank the anonymous reviewers of this paper, who provided many useful suggestions for its improvement.

This research was funded in part by NSF CNS-0524252.

References

[1] 107th U. S. Congress. Help America Vote Act of 2002. `http://www.fec.gov/hava/law_ext.txt`.

[2] Shuki Bruck, David Jefferson, and Ronald L. Rivest. A Modular Voting Architecture (“Frogs”). Workshop on Trustworthy Elections, 2001. `http://www.vote.caltech.edu/wote01/pdfs/amva.pdf`.

[3] Douglas W. Jones. Recommendations for the Conduct of Elections in Miami-Dade County using the ES&S iVotronic System. `http://www.cs.uiowa.edu/~jones/voting/miami.pdf`.

[4] Arthur Keller. Experiences with Sequoia AVC Edge with VeriVote Printer as Precinct Inspector in Santa Clara County. `http://gnosis.python-hosting.com/voting-project/June.2006/0081.html`.

[5] Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach. Analysis of an Electronic Voting System. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2004.

[6] Samuel Merrill. *Making Multicandidate Elections More Democratic*. Princeton University Press, 1988.

[7] David Molnar, Tadayoshi Kohno, Naveen Sastry, and David Wagner. Tamper-Evident, History-Independent, Subliminal-Free Data Structures on PROM Storage -or- How to Store Ballots on a Voting Machine. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[8] National Institute of Standards and Technology. FIPS 180-1: Secure Hash Standard. April 1995. `http://www.itl.nist.gov/fipspubs/fip180-1.htm`.

[9] Raymound S. Nickerson. Confirmation Bias: A Ubiquitous Phenomenon in Many Guises. *Review of General Psychology*, 2(2):175–220, 1998.

[10] Nokia. Python for Series 60. `http://www.forum.nokia.com/python`.

[11] Pygame. `http://pygame.org/`.

[12] Python Software Foundation. Python. `http://www.python.org/`.

[13] Markus Schulze. A New Monotonic and Clone-Independent Single-Winner Election Method. *Voting Matters*, (17):9–19, October 2003.

[14] T. Nicolaus Tideman. Independence of Clones as a Criterion for Voting Rules. *Social Choice and Welfare*, 4:185–206, 1987.

[15] U. S. Election Assistance Commission. *EAC 2004 Annual Report*. `http://www.eac.gov/docs/EAC%20Annual%20Report%20FY04.pdf`.