Report on the Pvote security review

Ka-Ping Yee
ping@zesty.ca

November 14, 2007

Contents

Chapter 1

Introduction

For three days in March 2007 and one day in May 2007, security researchers met at UC Berkeley to review the Pvote software for electronic voting machines. 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

All reviewers were given a hardcopy of the Pvote Software Review Assurance Document [3]. I was the author of this document and of the Pvote software. The line numbers mentioned in this report are as given in the source code listings in the assurance document (blank lines are not numbered). 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.

1.1 Overview

We 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. Our findings and recommendations concerning Pvote are described in Chapters 2 through 5. The discussion also generated many open questions and ideas about Pvote's design choices, which are summarized in Chapter 6.

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. The procedure and results of this test are detailed in Chapter 7.

Ideas concerning the software review process itself are described in Chapter 8. Chapter 9 reports on our informal post-review survey, and Chapter 10 highlights some of the lessons learned from the review.

1.2 Roles of the participants

The reviewers were tasked with evaluating the security of the design and implementation of Pvote.

David Wagner and I served as assistants to the reviewers, providing whatever information or tools they requested to aid them in their investigation. For the bug insertion exercise on the third day, David and I played the role of adversaries with respect to the reviewers, since we were specifically interested in the reviewers' ability to detect an insider attack.

1.3 Acknowledgements

I am extremely grateful to Matt Bishop, Ian Goldberg, Tadayoshi Kohno, Mark Miller, Dan Sandler, David Wagner, and Dan Wallach, who generously volunteered their time to participate in this security review.

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

Chapter 2

Correctness

This chapter describes the reviewers' findings on the question of whether Pvote met its stated security and correctness claims.

2.1 Pvote software

No bugs were found in the original Pvote 1.0 beta source code. Chapter 7 describes the bug-insertion experiment we conducted during the review.

2.2 Assurance document

The reviewers found some errors and omissions in the assurance document.

2.2.1 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.

2.2.2 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.

2.2.3 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.

2.2.4 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.

2.2.5 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.

2.2.6 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.

Chapter 3

Consensus recommendations

This chapter 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.

3.1 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.

3.2 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`.

3.3 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 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.

3.3.1 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.

3.4 Implementation

3.4.1 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**).

3.4.2 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.

3.4.3 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`.

3.4.4 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.

Chapter 4

Inconclusive recommendations

This chapter contains recommendations made during the review that did not reach general agreement, were disputed, or were ultimately withdrawn.

4.1 Ballot definition

4.1.1 Level of abstraction

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.

4.1.2 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).

4.2 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.

Chapter 5

Observations

This chapter documents other notable observations that reviewers made.

5.1 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.

5.2 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.

5.3 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.

5.4 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.

5.5 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.

5.6 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.

5.7 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).

5.8 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.

5.9 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.

5.10 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.

Chapter 6

Open issues

This chapter describes other unresolved issues and ideas that were discussed at the review concerning Pvote or software auditing in general.

6.1 Ballot definition

We discussed the following topics concerning the ballot definition.

6.1.1 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.

6.1.2 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.

6.2 Programming language

The effect of programming language design on source code review was another prominent topic.

6.2.1 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.

6.2.2 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.

6.2.3 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.

6.2.4 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.

6.2.5 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.

6.2.6 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.

6.2.7 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.)

6.3 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?

6.4 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 the audio, a cable running from the audio port to a recording device may go unnoticed [2].

6.5 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.

6.6 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.

6.7 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.

6.8 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.

6.9 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

6.10 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.

6.11 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.

Chapter 7

Bug insertion

This chapter 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 Pthin 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.

   ```
   if step.op == OP_REMOVE and selected:
      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.

   ```
   selections = self.selections[group_i]
   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.

   ```
   if option.writein_group_i != None:
      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.

7.1 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.

7.2 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 8.6), 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.

Chapter 8

Review process

This chapter describes ideas and suggestions regarding the software review process that came up during the review.

8.1 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.

8.2 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

8.3 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.

8.4 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.

8.5 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 commtited) 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.

8.6 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).

Chapter 9

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.

9.1 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.

9.2 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.

9.3 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.

9.4 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.

Chapter 10

Conclusions

This chapter highlights some of the lessons learned and themes that recurred throughout the review.

10.1 Conducting software reviews

Intentionally inserting bugs motivates reviewers. The bug-insertion experiment (see Chapter 7) 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.

10.2 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, such as the recursion in **Navigator.review()** and the use of invert in **Navigator.test()**. 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.

10.3 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 [1].

10.4 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.

Bibliography

[1] Mark S. Miller. Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control, 2006. http://erights.org/talks/thesis.

[2] Elliot Proebstel, Sean Riddle, Francis Hsu, Justin Cummins, Freddie Oakley, Tom Stanionis, and Matt Bishop. An Analysis of the Hart Intercivic DAU eSlate, 2007.

[3] Ka-Ping Yee. Pvote Software Review Assurance Document. UC Berkeley EECS Technical Report 2007-40, April 2007. http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-40.html.