Security Analysis of the Diebold AccuBasic Interpreter

David Wagner David Jefferson Matt Bishop
Voting Systems Technology Assessment Advisory Board (VSTAAB)

with the assistance of:

Chris Karlof Naveen Sastry
University of California, Berkeley

February 14, 2006

1 Summary

This report summarizes the results of our review of some of the source code for the Diebold AV-OS optical scan (version 1.96.6) and the Diebold AV-TSx touchscreen (version 4.6.4) voting machines. The study was prompted by two issues: (1) the fact that AccuBasic scripts associated with the AV-OS and AV-TSx had not been subjected to thorough testing and review by the Independent Testing Authorities when they reviewed the rest of the code for those systems, and (2) concern over vulnerabilities demonstrated in the AV-OS optical scan system by Finnish investigator Harri Hursti in Leon County, FL. Mr. Hursti showed that it is possible for someone with access to a removable memory card used with the AV-OS system to modify scripts (small programs written in Diebold's proprietary AccuBasic language) that are stored on the card, and also to modify the vote counts stored on the card, in such a way that the tampering would affect the outcome of the election and not be detected by the subsequent canvass procedures.

The questions we addressed are these:

- What kinds of damage can a malicious person do to undermine an election if he can arbitrarily modify the contents of a memory card?
- How can the possibility of such attacks be neutralized or ameliorated?

The scope of our investigation was basically limited to the above questions. We did not do a comprehensive code review of the whole codebase, nor look at a very broad range of potential security issues. Instead, we concentrated attention to the AccuBasic scripting language, its compiler, its interpreter, and other code related to potential security vulnerabilities associated with the memory cards.

We found a number of security vulnerabilities, detailed below. Although the vulnerabilities are serious, they are all easily fixable. Moreover, until the bugs are fixed, the risks can be mitigated through appropriate use procedures. Therefore, we believe the problems as a whole are manageable.

Our findings regarding the scope of possible attacks on the AV-OS optical scan and AV-TSx touchscreen systems can be summarized as follows:

- *AccuBasic is a limited language:* The AccuBasic language itself is not a powerful programming language, but a very restricted one, narrowly tailored to one task: calculating and printing reports before and after an election. From a security point of view this is very desirable; minimal functionality generally means fewer opportunities for error or security vulnerability. In particular, *when its interpreter is properly implemented* (see below) an AccuBasic program cannot modify votes or ballot images; it can read vote counters (AV-OS) or ballot images (AV-TSx), but it cannot modify them.

- *The AccuBasic interpreter is well-structured:* The code in the AccuBasic interpreters for both machines is clean, well-structured, and internally documented. We were able to understand it with little difficulty despite the lack of external documentation.

- *Memory card attacks are a real threat:* We determined that anyone who has access to a memory card of the AV-OS, and can tamper it (i.e. modify its contents), and can have the modified cards used in a voting machine during election, can indeed modify the election results from that machine in a number of ways. The fact that the the results are incorrect cannot be detected except by a recount of the original paper ballots.

- *Harri Hursti's attack does work:* Mr. Hursti's attack on the AV-OS is definitely real. He was indeed able to change the election results by doing nothing more than modifying the contents of a memory card. He needed no passwords, no cryptographic keys, and no access to any other part of the voting system, including the GEMS election management server.

- *Interpreter bugs lead to another, more dangerous family of vulnerabilities:* However, there is another category of more serious vulnerabilities we discovered that go well beyond what Mr. Hursti demonstrated, and yet require no more access to the voting system than he had. These vulnerabilities are consequences of bugs—16 in all—in the implementation of the AccuBasic interpreter for the AV-OS. These bugs would have no effect at all in the absence of deliberate tampering, and would not be discovered by any amount of functionality testing; but they could allow an attacker to completely control the behavior of the AV-OS. An attacker could change vote totals, modify reports, change the names of candidates, change the races being voted on, or insert his own code into the running firmware of the machine.

- *Successful attacks can only be detected by examining the paper ballots:* There would be no way to know that any of these attacks occurred; the canvass procedure would not detect any anomalies, and would just produce incorrect results. The only way to detect and correct the problem would be by recount of the original paper ballots, e.g. during the 1 percent manual recount.

- *The bugs are classic, and can only be found by source code review:* Finding these bugs was only possible through close study of the source code. All of them are classic security flaws, including buffer overruns, array bounds violations, double-free errors, format string vulnerabilities, and several others. There may, of course, be additional bugs, or kinds of bugs, that we did not find.

- *AV-TSx has potential cryptographic protection against memory card attacks:* A majority of the bugs in the AV-OS AccuBasic interpreter are also present in the interpreter for the AV-TSx touchscreen system. However, the AV-TSx touchscreen has an important protection that the AV-OS optical scan does not: the key contents of its removable memory card, including the AccuBasic scripts, are digitally signed. Hence, if the cryptographic keys are managed properly (see next bullet), any tampering would be quickly detected and the attack would be unsuccessful. All of the attacks we describe, and Hursti's attack as well, would be foiled, because the memory card by itself would in effect be cryptographically tamperproof.

- *But the implementation of cryptographic protection is flawed:* There is a serious flaw in the key management of the crypto code that otherwise should protect the AV-TSx from memory card attacks. Unless election officials avail themselves of the option to create new cryptographic keys, the AV-TSx uses a default key. This key is hard-coded into the source code for the AV-TSx, which is poor security practice because, among other things, it means the same key is used in every such machine in the U.S. Worse, the particular default key in question was openly published two and a half years ago in a famous research paper, and is now known by anyone who follows election security, and can be found through Google. The result is that in any jurisdiction that uses the default keys rather than creating new ones, the digital signatures provide no protection at all.

- *All the bugs are easy to fix:* In spite of the fact that the bugs we have identified are very serious, all of them are very local and very easy to fix. In each case only a couple of lines of code need to be changed. It should take only a few hours to do the whole job for both the AV-OS and AV-TSx.

- *No use of high assurance development methods:* The AccuBasic interpreter does not appear to have been written using high-assurance development methodologies. It seems to have been written according to ordinary commercial practices. In the long run, if the interpreter remains part of the codebase, it and the rest of the codebase should be revised according to a more rigorous methodology that would, among other things, likely have prevented the bugs we found.

- *Interpreted code is contrary to standards:* Interpreted code in general is prohibited by the 2002 FEC Voluntary Voting System Standards, and also by the successor standard, the EAC's Voluntary Voting System Guidelines due to take effect in two years. In order for the Diebold software architecture to be in compliance, it would appear that either the AccuBasic language and interpreter have to be removed, or the standard will have to be changed.

- *Bugs detailed in confidential companion report:* In a companion report we have listed in great detail all of the bugs we identified, the lines at which they occur, and the threats they pose. Because that report contains Diebold proprietary information, and because it details exactly how to exploit the vulnerabilities we discovered, that report must be confidential.

Clearly there are serious security flaws in current state of the AV-OS and AV-TSx software. However, despite these serious vulnerabilities, we believe that the security issues are manageable by a reasonably careful combination of short- and long-term approaches. Here are our recommendations with regard to mitigation strategies.

In the short term, especially for local elections, the security problems related to AccuBasic and the memory cards might be managed according to guidelines such as these:

- *Strong control over access to memory cards for the AV-OS:* The AV-OS optical scan is vulnerable to both the Hursti attack and attacks based on the AccuBasic interpreter bugs we found. It would be safest if it is not widely used until these bugs are fixed, and until a modification is made to ensure that the Hursti attack is eliminated. But if the AV-OS is used, strong procedural safeguards should be implemented that prevent anyone from gaining unsupervised or undocumented access to a memory card, and these procedures should be maintained for the life of all cards. Such controls might include a dual-person rule (i.e. no one can be alone with a memory card); permanent serial numbers on memory cards along with chain-of custody documentation, so there is a paper trail to record who has access to which cards; numbered, tamper evident seals protecting access to the cards whenever they are out of control of county staff; and training of all personnel, including poll workers, regarding proper treatment of cards, and how to check for problems with the seals and record a problem. Any breach of control over a card should require that its contents be zeroed (in the presence of two people) before it is used again.

- *Require generation of new crypto keys for the AV-TSx:* The AV-TSx is not vulnerable to any of these memory card attacks *provided* that the default cryptographic key used for signing the contents of the memory card is changed to a new, unguessable key and kept secure. If the key is changed then these threats are all eliminated, at least for the short term. If this is not done, however, then the AV-TSx is no more secure than the AV-OS.

- *Control access to GEMS:* Access to GEMS should be tightly controlled. This is a good idea for many reasons, since a malicious person with access to GEMS can undermine the integrity of an election in many ways. In addition, in a TSx system, GEMS holds a copy of the cryptographic key used for signing the contents of the memory cards, and in both systems the GEMS server may hold master copies of the AccuBasic scripts loaded onto the memory cards.

In the longer term, one would want to consider a number of additional measures:

- *Fix bugs:* Certainly the bugs in the source code of the interpreters for both the AV-OS and AV-TSx should be corrected with all deliberate speed, the Hursti vulnerability should be fixed, and the code re-examined by independent experts to verify that it was properly done.

- *Defensive and high assurance programming methodology:* The source code of the interpreters should be revised to introduce systematic defensive programming practices and high assurance development methods. In particular, eliminate in the firmware, insofar as possible, any trust of the contents of the memory card.

- *Protect AccuBasic code from tampering:* The AccuBasic object code could be protected from tampering and modification, either by (a) storing AccuBasic object code on non-removable storage and treating it like firmware, or by (b) protecting AccuBasic object code from modification through the use of strong cryptography (particularly public- key signatures).

- *Don't store code on memory cards:* The architecture of the AV-OS and the AV-TSx could be changed so they do not store code on removable memory cards.

- *Remove interpreters and interpreted code:* The architecture of the AV-OS and the AV-TSx could be changed so they do not contain any interpreter or use any kind of interpreted code, in order to bring the codebase into compliance with standards.

2 Introduction

Scope of the study. This report summarizes the results of our review of the source code for the Diebold AV-OS optical scan (version 1.96.6) and the Diebold AV-TSx touchscreen (version 4.6.4) voting machines. This investigation, requested by the office of the California Secretary of State, was to evaluate security concerns raised by the use of AccuBasic scripts (programs) stored on removable memory cards in the two systems and offer options for their amelioration. The study was prompted by vulnerabilities demonstrated in the optical scan system by Finnish investigator Harri Hursti in Leon County, FL. Mr. Hursti showed that under certain circumstances it is possible for someone with access to a memory card to modify the scripts and modify the vote counts in a way that would not be detected by the subsequent canvass procedure, and would normally only be detectable by a recount of the paper ballots.

Our study does not constitute a comprehensive code review of the entire Diebold codebase. We had access to the full codebases for the AV-OS and AV-TSx, but we did not even attempt a comprehensive review of the entire codebase. Our attention was focused fairly narrowly on Diebold's proprietary AccuBasic scripting language, the compiler for that language, the interpreter for its object code, the AccuBasic scripts themselves, and the related protocols and procedures, both for the AV-OS (optical scan) and AV-TSx (touchscreen) voting systems.

In particular, we did not have the source code for the Diebold GEMS election management system, and our security evaluation does not cover GEMS at all. It is widely acknowledged that a malicious person with unsupervised access to GEMS, even without knowing the passwords, can compromise GEMS and the election it controls. This report does not address those threats, however.

Our analysis was based only on reading the source code we were given. We did not have access to a real running system (although we were able to compile and execute modified versions of the compiler and interpreter on a PC). Nor did we have any manuals or other documentation beyond that present in comments in the code itself. We had access to the source code for a period of approximately four weeks for this review.

The threat model. Different jurisdictions around the country have somewhat different procedures for conducting an election with the Diebold AV-OS and AV-TSx systems, but all include the following steps:

1. Before the election, the removable memory cards are initialized though the GEMS election management system with the appropriate election description information for the precinct the machine will be used in, and with the AccuBasic object code scripts to be used, and with other information detailed below.

2. The initialized cards are then inserted into the voting machines (optical scan or touchscreen); the compartment in which the card sits is locked and sealed with a tamper-evident seal of some kind.

3. The voting machine with its enclosed card is transported to the precinct poll site where it is stored over night (or longer) until the start of the election.

4. At the start of the election, a script on the card is used to print initial reports, including the Zero Report, which should indicate that all the vote counters are zero (in the AV-OS) and file of voted ballots is empty (in the AV-TSx).

5. All during election day, voted paper ballots are scanned and the appropriate counters on the removable memory card are incremented (AV-OS), or the voted ballots themselves are stored electronically on the memory card (AV-TSx), and electronic audit log records are appended to a file on the card.

6. At the end of election day, a script from the card is used to print final reports for the day, including vote totals.

7. Finally, one of two steps is taken, depending on the jurisdiction: either (a) the seal is broken and the memory card is removed and transported back to a central location for canvass using GEMS; or, (b) the entire voting machine is transported to the central location, where election officials break the seal, remove the memory card, and read its contents during the canvass.

The threats we are concerned about specifically involve modification of the contents of the memory card, especially the AccuBasic object code. In other words, somewhere along the line, in the procedure above, the attacker is able to get a memory card, arbitrarily modify its contents, and surreptitiously place it in a voting machine for use in an election, and do so without being immediately detected.

We assume the attacker's goal is either to change the election results undetected, or perhaps simply to disrupt the election (e.g. by causing voting machine crashes). We also assume that the attacker knows every detail of how the system works, and the procedural safeguards, and even has access to the manuals, documentation, and source code of the system. The attacker, therefore, is able to take advantage of bugs and vulnerabilities in the code. (It is standard to make these last assumptions, since it is almost impossible to keep code and related information secret from a determined attacker.)

We do not, however, assume that the attacker has any inside confederates, or has access to any passwords or cryptographic keys, or access to GEMS. We do not assume that the attacker has any access to paper ballots (AV-OS) or VVPAT (AV-TSx), nor even that he has any access to the voting system beyond the ability to insert a memory card undetected.

The process we followed. We were asked to perform a security review of the Diebold source code. As part of the review, we were provided access to the source code for the AV-OS and the AV-TSx machines. This included the source code for the AccuBasic compiler, for the AccuBasic interpreter in the AV-OS and the AccuBasic interpreter in the AV-TSx, for some AccuBasic scripts, and all other source code for the AV-OS and AV-TSx. There are two separate versions of the interpreter, one in the AV-OS and one in the AV-TSx; however, the two implementations are very similar.

We undertook a line-by-line analysis of the source code for the AV-OS AccuBasic interpreter. Three team members (Karlof, Sastry, and Wagner) read every line of source code carefully and checked for all types of security and reliability defects known to us. When we found a vulnerability in the AV-OS interpreter, we examined the corresponding portion of the AV-TSx interpreter to check whether the AV-TSx shared that same vulnerability.

After completing the line-by-line source code analysis, we applied a commercial static source code analysis tool to the AV-OS interpreter code. Code analysis tools perform an automated scan of the source code to identify potentially dangerous constructs. We obtained a copy of the Source Code Analyzer (SCA) tool, made by Fortify Software, Inc.; Fortify generously donated the tool to us for our use in this project at no cost, and we gratefully acknowledge their contribution. Two of us (Bishop and Wagner) are members of Fortify Software's Technical Advisory Board, and thus were already familiar with this tool. We manually inspected each of the warnings generated by the tool.

While our analysis uncovered several potential attacks on the system, we have not attempted to attack any working system. We performed our analysis mostly "on paper"; we did not have access to a genuine running system. We did, however, get a stubbed-out version of the code running on a PC, and were able to confirm that one of the attacks we discovered (the only one we tried) actually works.

In the end, we wrote our report in two parts. The *public* part is this document, which contains background, our findings and recommendations, and all of the explanatory information we have found to support them. The *confidential* part contains a detailed description of all of the bugs we found, the file names and line numbers where they occur, how they can be exploited, and what the consequences are. It is confidential because it contains both proprietary material and specific information about potential attacks on voting systems.

3 Background

3.1 Contents of the memory card

Both the AV-OS and AV-TSx systems use removable memory cards as key parts of their architectures. In both systems, the memory cards contain several kinds of information:

- the election description (a small database describing the races, candidates, parties, propositions, and ballot layout information for the current election);
- vote counters for every candidate and proposition on the ballot that store a count of the number of votes for that candidate (in the case of the AV-OS), or data records containing the cast ballot images (AV-TSx), along with various summary counters;
- byte-coded object programs (.abo files), which are normally created by writing scripts (programs) in the AccuBasic language and running them through the AccuBasic compiler.[1]
- the internal electronic audit log
- an election mode field indicating whether the system containing the card is currently being used in a real election or not;
- a large number of other significant variables including strings, flags (for selecting options), various event counters, and other data describing the state of the election.

In fact, as far as we can tell, *the entire election-specific state of the voting machine* (the part that is retained *between* voting transactions) is stored on the memory card. It would take a much more comprehensive review of the software than we were able to conduct in order to verify this, but it appears to be the case.

[1]AccuBasic object files (.abo files) are *normally* created by running AccuBasic programs through the compiler, i.e. that is the intent. But nothing prevents a programmer from directly writing .abo files, or modifying them, bypassing the AccuBasic language and the compiler entirely. Indeed, this is a route to several potential attacks. The AccuBasic interpreter makes no effort to verify that the AccuBasic object code has indeed been produced by the compiler.

All of this information on the memory cards is critical election information. If it is not properly managed, or if it is modified in any unauthorized way, the integrity of the entire election is possibly compromised. It is therefore vital, as everyone acknowledges, to maintain proper procedural control over the memory cards to prevent unauthorized tampering, and to treat them at all times during the election *with at least the same level of security as ballot boxes containing voted ballots.*

From one point of view, such an architecture makes good sense. In principle, it allows a memory card to be removed from a machine at almost any time (except during a short critical time window at the final completion of each vote transaction) without losing any votes or audit records, or any of the other context that has been accumulated. (Removal of a memory card during an election is procedurally forbidden under normal circumstances.) And it guarantees that when the memory card is removed at the end of the day, it contains *all* of the data needed for canvass, and for the resolution of most disputes, excepting only those that might depend on detailed forensic analysis.

Having all of the state on a removable memory card has a downside, however. It means an attacker with access to the card has potentially many other avenues of attack besides direct modification of the vote counts or the AccuBasic scripts; he can modify any other part of the election configuration or state as well. In our investigation, we did not attempt to enumerate all of these possibilities since it was clear that the only strong way to protect against all such attacks is to prevent any possibility of undetected tampering with the memory card in the first place.

When the AV-OS memory card is inserted into the AV-OS, it acts like an extension of main memory, and can be directly read and written via ordinary memory addressing, e.g. via variables and pointers. (Whether it actually is RAM, or is instead some other kind of memory-mapped storage device is not clear to us, but from a software point of view there is no difference.)

On the AV-TSx, however, the election state data is stored in a *file system* on the removable card. This means that the firmware cannot access it directly as main memory, but must use open/close/read/write calls to move data between files on the card and main memory. From a reliability and security point of view this is preferable to the architecture used on the AV-OS, since many kinds of common bugs (e.g. index or pointer bugs) can corrupt the data on a card that acts as main memory, whereas that is less likely for data packaged in a file system.

In the AV-OS, once the memory card is inserted into the voting machine, the byte-coded object programs become immediately executable by the AccuBasic interpreter in the firmware of the machine. However, on the AV-TSx the byte-coded object programs are cryptographically protected by the GEMS election management system. In effect, the GEMS server writes a sort of checksum[2] that depends on both the data and a secret cryptographic key to the memory card. When the memory card is inserted in an AV-TSx machine, the correctness of the checksum is validated and the machine refuses to enter election mode if the check fails[3].

The cryptographic protection for the object code on the AV-TSx touchscreen machine is a significant improvement. It means that even if an attacker can get access to a memory card and modify the object code, unless he also has the cryptographic key to allow him to create a matching checksum for the modified object code, the checksum will not match when the card is inserted and the attack would be foiled. The integrity of the object code then boils down, for all practical purposes, to the secrecy of the cryptographic key (which we will discuss later).

[2]To be precise, it uses a cryptographic message authentication code (MAC).

[3]If the cryptographic message authentication code is invalid, a dialog box appears on the screen with the warning "Unable to load the election: the digital database signature does not match the expected value", and the machine does not enter election mode.

3.2 AccuBasic

The AccuBasic programming language is a Diebold-proprietary, limited-functionality *scripting language* (a kind of programming language). The *scripts* (programs) written in AccuBasic are intended to be used only for creating and printing reports on the printer units attached to the AV-OS or AV-TSx.

Once a script is written in AccuBasic (the *source code* version of the script), it is run through the AccuBasic *compiler*, which translates it into a form of *object code*. The object code is represented in another Diebold-proprietary language that seems to be unnamed but is generally referred to as *byte code* or an *.abo file*. It is the object code form of the scripts that is stored on the memory card, not the source form.

Normally all .abo files are produced in this way, i.e. by running AccuBasic source through the compiler. But it is important to understand that nothing prevents a programmer from bypassing the compiler and constructing a valid .abo file directly, or by editing an .abo file produced by the compiler. (Mr. Hursti did just that, modifying the portion of the script responsible for printing the zero report.) A .abo file produced in either of these nonstandard ways might not be producible by the compiler at all from any AccuBasic source file. However, they will still be executable by the interpreter without any error, and this fact can be the basis for powerful attacks that can take advantage of bugs in the interpreter. The AccuBasic interpreter makes no attempt to validate the .abo files, i.e. to ascertain that that they were in fact produced using the compiler.

The AccuBasic software for the AV-TSx is slightly different from that on the AV-OS. This is due primarily to the differences in the environment on the two systems. For example, the AV-TSx gets yes/no user input through the touchscreen, whereas the AV-OS gets it from physical buttons. Also, AV-OS memory cards contain vote counters only, whereas the AV-TSx cards store full ballot records. The memory card on the AV-OS is memory-mapped, whereas the same information is stored in a file system on the AV-TSx memory card. The AccuBasic interpreter for the AV-TSx is implemented in C++, whereas the interpreter in the AV-OS is written in C. The AV-OS interpreter contains 1838 lines of C code (not counting blank lines, comments, or global declarations), while the AV-TSx contains 2614 lines of C++ code (again, excluding blank lines, comments, and declarations). However, it is clear that the AccuBasic interpreter in the AV-TSx was originally just a translation from C to C++ of the one in the AV-OS, and they have subsequently diverged only slightly. The differences between the two AccuBasic interpreters are generally small enough that, except where noted, our generalizations about AccuBasic and its implementation apply equally to both versions.

AccuBasic is in one sense a general purpose language, in that it is able to do arbitrary numerical and string calculations.[4] But in another sense, *when its interpreter is properly implemented*, it is a very restricted language in that, while it can *calculate* anything, it can only *control* a very limited part of the functionality of the voting machine. For example, an AccuBasic script can read the vote counters (or ballot images) and the election description from the memory card, and it can read a few other internal values as well (such as the date and time); but it cannot modify any of them. And it can invoke only a few functions from the rest of the codebase outside the interpreter, specifically, those needed for assembling information for, and for the printing of, reports on the machine's screen and printer. It is not possible (again, *when the AccuBasic interpreter is properly implemented*) for AccuBasic object code to:

[4]The language uses integer and string data types, and permits assignments, substring extraction and assignment, conditionals, loops, a limited number of defined subroutines, subroutine calls (without arguments), and recursion. It is theoretically capable of computing any computable function.

- modify the vote counts (AV-OS) or the ballot images (AV-TSx);
- forge any votes or fail to record any votes;
- modify the election description information; or
- modify any paper ballots.

On the other hand, even when perfectly implemented, it is always possible for an erroneous or malicious AccuBasic script to:

- print false reports, or
- crash the voting machine (e.g., by going into an infinite loop).

These latter points are not flaws in the design of AccuBasic language or interpreter. Any other software, e.g. the machine's firmware, could have similar bugs. However, the fact that the scripts are on removable memory cards—and thus potentially exposed to tampering—makes these possibilities important. Mr. Hursti's attack on the AV-OS depended critically on his ability to modify the Zero Report script so that it falsely indicated that all counters were zero when in fact they were not. And in some jurisdictions, e.g. Florida, the reports printed by the AV-OS are the legal results of the election, so printing a false report amounts to falsifying the results of the election.

The intent of the AccuBasic language, compiler, and interpreter is that AccuBasic scripts should be usable *exclusively* for creating and printing reports on the voting machine's printer, without modifying the voting machine's behavior in any other way. With the exception of some serious bugs (described in our findings below) we found that this is indeed the case. In spite of its name, which is reminiscent of the powerful scripting language Visual Basic, we found that AccuBasic is a very limited, special purpose language; this is the right approach if one is to use an interpreted language at all.

Aside from the bugs (described below) the AccuBasic interpreters for both the AV-OS and AV-TSx are very well written and documented. We had no difficulty understanding the code and reviewing it.

4 Findings

Finding 1 *There are serious vulnerabilities in the AV-OS and AV-TSx interpreter that go beyond what was previously known. If a malicious individual gets unsupervised access to a memory card, he or she could potentially exploit these vulnerabilities to modify the electronic tallies at will, change the running code on these systems, and compromise the integrity of the election arbitrarily. (The original paper ballots for the AV-OS, of course, cannot be affected by tampering with the memory cards.)*

The AccuBasic interpreters, in both the AV-OS and AV-TSx, have a number of serious bugs—defects in the source code—that render the machines vulnerable to various attacks. (This goes well beyond what Mr. Hursti demonstrated; his attacks did not exploit any of these vulnerabilities.) These vulnerabilities would not affect the normal behavior of the machine, and would not be discovered during testing. But they could be exploited by an attacker with unsupervised access to a memory card. Many of these vulnerabilities are present in both the AV-OS and AV-TSx;

the AV-TSx code is basically a translation of the AV-OS code from C to C++, and most of the vulnerabilities were preserved in the translation.

The vulnerabilities arise because the AccuBasic interpreter "trusts" the contents of the AccuBasic object code (.abo files) stored on the memory card, and implicitly assumes that this AccuBasic object code has been produced by a legitimate Diebold AccuBasic compiler. As discussed earlier, this assumption is not necessarily justified. Anyone with unsupervised access to the AV-OS memory card could freely modify its contents, including the .abo file stored on the memory card. The same is true of the AV-TSx memory card, if the cryptographic keys are not updated from their default values (see Finding 4 below).

Types of vulnerabilities. The vulnerabilities include several instances of the classic buffer overrun vulnerability, as well as vulnerabilities with a similar effect. This kind of vulnerability would allow someone who could edit the AccuBasic object code on the memory card to completely control the behavior of the voting machine. The instant that the AccuBasic interpreter on the AV-OS or AV-TSx attempts to execute the malicious AccuBasic object code, the machine will be compromised.

Table 1 contains an overview of the 16 vulnerabilities we found in the AV-OS, and their impact. Also, Table 2 contains a similar overview of the 10 vulnerabilities we found in the AV-TSx. Note that we have excised any information that might help to exploit these vulnerabilities from those tables. We have relegated all such information to a separate Appendix, which contains additional detail: for each vulnerability, the Appendix lists the source code line number where the vulnerability appears, along with information about how the vulnerability might be exploited in the field.

These vulnerabilities were found primarily by line-by-line review of the source code, performed by three of us reading every line of the interpreter code together as a team. After we had completed a careful line-by-line security analysis, we then applied the Fortify Source Code Analyzer (SCA) tool and examined the warnings it produced. Given the care with which we performed the manual code review, we had not expected a static bug-finding tool to find any further bugs. Consistent with our expectations, the first warning we inspected from the tool referred to an exploitable security vulnerability we had already found. However, to our considerable surprise, the second warning from the tool turned out to reveal a vulnerability that we had missed as part of our manual code inspection (namely, Vulnerability V2). (The remainder of warnings we examined pointed to bugs and vulnerabilities that we had already found.)

In all cases the specific bugs we found are local and easy to fix. One concern, however, is that these are just the bugs *we* were able to find; there are quite possibly others we did not notice, and that automated bug-finding tools (which are always imperfect) would not notice either. Code review is difficult. It is hard to be confident that one has found all bugs (and indeed, our experience with the Fortify SCA tool highlighted this fact), and if we used another tool or if another person were to examine the code, they might find other vulnerabilities.

None of the vulnerabilities we found would have been found through standard testing, so testing is not the answer. This is a long-term problem with the use of interpreted code on removable memory cards, and with the failure to use defensive programming and other good security practices when implementing the interpreter.

These vulnerabilities have not been confirmed by verifying that they work against a full working system. (We did not have access to a running system.) We have used our best judgement to assess which bugs are likely to be exploitable, but it is possible that some bugs we classified as vulnerabilities may in fact not be exploitable. Conversely, there may be other vulnerabilities that we failed to identify because of the lack of a working system.

To double-check our analysis, we chose one vulnerability more or less at random and verified that we were able to exploit it in a simulated test environment. We were able to compile and execute a slightly modified version of the AV-OS AccuBasic interpreter, as well as the AccuBasic compiler, on a PC. We then developed an example of AccuBasic object code (an .abo file) that would exploit this vulnerability. We verified that, when using the interpreter to interpret this object code on our PC, we were able to trigger a buffer overrun and successfully exploit the vulnerability. This provides partial confirmation of our analysis, but it is certainly not an authoritative test. We did not attempt to perform an exhaustive test of all 16 vulnerabilities.

Impact. The consequence of these vulnerabilities is that any person with unsupervised access to a memory card for sufficient time to modify it, or who is in a position to switch a malicious memory card for a good one, has the opportunity to completely compromise the integrity of the electronic tallies from the machine using that card.

Many of these vulnerabilities allow the attacker to seize control of the machine. In particular, they can be used to replace some of the software and the firmware on the machine with code of the attacker's choosing. At that point, the voting system is no longer running the code from the vendor, but is instead running illegitimate code from the attacker. Once the attacker can replace the running code of the machine, the attacker has full control over all operation of the machine. Some of the consequences of this kind of compromise could include:

- The attack could manipulate the electronic tallies in any way desired. These manipulations could be performed at any point during the day. They could be performed selectively, based on knowledge about running tallies during the day. For instance, the attack code could wait until the end of the day, look at the electronic tallies accumulated so far, and choose to modify them only if they are not consistent with the attacker's desired outcome.
- The attack could print fraudulent zero reports and summary reports to prevent detection.
- The attack could modify the contents of the memory card in any way, including tampering with the electronic vote counts and electronic ballot images stored on the card.
- The attack could erase all traces of the attack to prevent anyone from detecting the attack after the fact. For instance, once the attack code has gained control, it could overwrite the malicious AccuBasic object code (.abo file) stored on the memory card with legitimate AccuBasic object code, so that no amount of subsequent forensic investigation will uncover any evidence of the compromise.
- It is even conceivable that there is a way to exploit these vulnerabilities so that changes could persist from one election to another. For instance, if the firmware or software resident on the machine can be modified or updated by running code, then the attack might be able to modify the firmware or software in a permanent way, affecting future elections as well as the current election. In other words, these vulnerabilities mean that a procedural lapse in one election could potentially affect the integrity of a subsequent election. However, we would not be able to verify or refute this possibility without experimentation with real systems.

	Type	Impact
V1	Array bounds violation	Overwrite any memory address within $\pm 2^{15}$ bytes of the global context structure with a 2-byte value that the adversary has partial control over. Might allow attacker to inject malicious code and take complete control of the machine. Might allow overwriting vote counters.
V2	Format string vulnerability	Crash the machine; read the contents of memory within a narrow range
V3	Input validation error	Choose any location on the memory card and begin executing it as .abo code; could be used to conceal malicious .abo code in unexpected locations, or to crash the machine.
V4	Array bounds violation	Memory corruption; crash the machine.
V5	Double-free() vulnerability	Overwrite any desired 4-byte memory address with any desired 4-byte value. Allows attacker to inject malicious code and take complete control of the machine.
V6	Array bounds violation	Memory corruption: overwrite any memory address up to 2^{16} bytes after the global context structure with a 2-byte value that the adversary has no control over. Might allow overwriting vote counters.
V7	Buffer overrun	Memory corruption; crash the machine
V8	Buffer overrun, integer conversion bug	Memory corruption: overwrite up to 2^{15} consecutive bytes of memory starting at global context structure. Might allow attacker to inject malicious code and take complete control of the machine. Might allow overwriting vote counters. Information disclosure: read any memory location $\pm 2^{15}$ bytes away from global context structure. Crash the machine.
V9	Buffer underrun	Memory corruption: overwrite up to 2^{15} consecutive bytes of memory extending backwards from the global context structure. Might allow attacker to inject malicious code and take complete control of the machine. Might allow overwriting vote counters. Information disclosure: read any memory location within this window. Crash the machine.
V10	Buffer overrun	Overwrite return address on the stack. Allows attacker to inject malicious code and take complete control of the machine.
V11	Array bounds violation	Information disclosure: read from potentially any memory address. Crash the machine.
V12	Array bounds violation	Write any 2-byte value to any address up to 2^{16} bytes after the global context structure. Might allow attacker to inject malicious code and take complete control of the machine. Might allow overwriting vote counters.
V13	Array bounds violation	Information disclosure: Read any 2-byte value from any address up to 2^{16} bytes after the global context structure.
V14	Pointer arithmetic error	Crash machine. Could begin interpreting random memory locations as though they were .abo code.
V15	Unchecked string operation	Machine might crash or become unresponsive
V16	Unchecked string operation	Overwrite stack memory. Might allow attacker to inject malicious code and take complete control of the machine.

Table 1: 16 security vulnerabilities we found in the AV-OS.

	Type	Impact
W1	Array bounds violation	Overwrite any memory address with a 4-byte value that the adversary has partial control over. Allows attacker to inject malicious code and take complete control of the machine.
W3	Input validation error	Choose any memory location and begin executing it as .abo code; could be used to conceal malicious .abo code in unexpected locations, or to crash the machine.
W6	Array bounds violation	Overwrite any memory location with any desired value. Allows attacker to inject malicious code and take complete control of the machine.
W7	Buffer overrun	Memory corruption; crash the machine
W8	Buffer overrun, integer conversion bug	Corrupts memory until the machine crashes.
W10	Buffer overrun	Overwrite return address on the stack. Allows attacker to inject malicious code and take complete control of the machine.
W11	Array bounds violation	Information disclosure: read from potentially any memory address. Crash the machine.
W12	Array bounds violation	Writes any 4-byte value to any address. Allows attacker to inject malicious code and take complete control of the machine.
W13	Array bounds violation	Information disclosure: read a 4-byte value from any address.
W14	Pointer arithmetic error	Crash machine. Could begin interpreting random memory locations as though they were .abo code.

Table 2: 10 security vulnerabilities we found in the AV-TSx. Note that in many cases, the same vulnerability appears in both the AV-OS and AV-TSx interpreters, so we have used parallel numbering (e.g., the bug V6 in the AV-OS interpreter also appears in a very similar form as bug W6 in the AV-TSx interpreter).

- It is conceivable that the attack might be able to propagate from machine to machine, like a computer virus. For instance, if an uninfected memory card is inserted into an infected voting machine, then the compromised voting machine could replace the AccuBasic object code on that memory card with a malicious AccuBasic script. At that point, the memory card has been infected, and if it is ever inserted into a second uninfected machine, the second machine will become infected as soon as it runs the AccuBasic script.

  It is difficult to confidently assess the magnitude of this risk without experimentation with real systems. That said, given our current understanding of how memory cards are used and our current understanding of the vulnerabilities[5], we believe the risk of this kind of attack is low (at least in the near term). This kind of virus would only be able to spread through "promiscuous sharing" of memory cards, which means that propagation would probably be fairly slow. If typical practice is that memory cards are wiped clean before the election, programmed, sent to the polls, and then returned for reading at the GEMS central management system, then there does not seem to be much opportunity for one infected memory card to infect many machines.

- On the AV-TSx, the attack could print fraudulent VVPAT records. Since VVPAT records are considered the authoritative record during a recount, this might enable election fraud even if the VVPAT records are manually recounted. For instance, the attack could print extra VVPAT records during a quiet time when no voter is present (however, we expect that this might be noticed by poll workers, as the TSx printer is fairly noisy). As another example, when a voter is ready to print the VVPAT record, the attack code could print two copies of the voter's VVPAT record and hope that the voter doesn't notice. The attack might print duplicate VVPAT records only for voters who have voted for one particular candidate, thereby inflating the number of VVPAT records for that favored candidate. Alternatively, it might fail to print VVPAT records for voters who vote for a disfavored candidate (but of course, this could easily be detected voters who know to expect the machine to print a VVPAT record).

  We believe the risk of false VVPAT records is lower than it might at first seem. See below for further discussion.

- The attack could affect the correct operation of the machine. For instance, on the AV-OS, it could turn off under- and over-vote notification. It could selectively disable over-vote notification for ballots that contain votes for a disfavored candidate, or selectively provide false over-vote notifications for ballots that contain votes for a favored candidate. On the AV-TSx, it could show the voter a wrong or incomplete list of candidates during vote selection; it could change selections between the time when they are initially selected and when they are shown on the summary screen; and it could selectively target a subset of voters, based on how they have voted or on other factors. Once the machine is running native code supplied by the attacker, its operation can be completely controlled by the attacker.

In addition, most of the bugs we found could be used to crash the machine. This might disenfranchise voters or cause long lines. These bugs could be used to selectively trigger a crash

[5]We have assumed as part of this analysis that the GEMS central management system, and TSx machines running in accumulator mode, do not execute AccuBasic scripts as part of reading memory cards. We were not able to verify or refute this assumption; however, we have no reason to believe it is inaccurate. Of course, if this assumption is inaccurate, our analysis of the risk would be affected.

only on some machines, in some geographic areas, or based on certain conditions, such as which candidate has received more votes. For instance, it would be possible to write a malicious AccuBasic script so that, when the operator prints a summary report at the end of the day, the script examines the vote counters and either crashes or continues operating normally according to which candidate is in the lead.

Unfortunately, the ability of malicious AccuBasic scripts to crash the machine is currently embedded in the architecture of the interpreter. Any infinite loop in the AccuBasic script immediately translates into an infinite loop in the interpreter (which causes the machine to stop responding, and is indistinguishable from a crash), and any infinite recursion in the AccuBasic script translates into stack overflow in the interpreter (which could corrupt stack memory or crash the machine).

The impact on the paper ballots (AV-OS). It is important to note that even in the worst case, the paper ballots cast using an AV-OS remain trustworthy; in no case can any of these vulnerabilities be used to tamper with the paper ballots themselves.

The impact on the VVPAT records (AV-TSx). As mentioned above, on the AV-TSx it is conceivable that these vulnerabilities might enable an attacker to print false VVPAT records. We assess the magnitude of this risk here. There are two cases:

- If the bugs are not fixed, and if proper cryptographic defenses are not adopted (see Finding 3), and if a malicious individual gains unsupervised access to the memory code:

  > In this case, it is hard to make any guarantees about the integrity of the VVPAT records. Attack code might be able to introduce fraudulent VVPAT records, compromising the integrity of both the electronic tallies and the paper records.
  >
  > We were unable to identify any realistic scenario where this would enable an attacker to cause fraud on a large enough scale to affect the outcome of a typical election without being detected. If the attack tries to insert many fraudulent extra VVPAT records, then the 1% recount should detect that the VVPAT records do not match the electronic tallies or that many precincts have more VVPAT records than voters who signed in (on the roster sheets), which would reveal the presence of some kind of attack and (presumably) trigger further investigation. If the attack tries to defraud many voters by failing to print a valid VVPAT record, then we suspect at least some of these voters will notice and the attack is likely to be detected. Also, mounting a large-scale attack would appear to require tampering with many memory cards or with the GEMS election management system, which restricts the class of adversaries who would have the opportunity to mount such an attack.
  >
  > Nonetheless, if such an attack is detected, it may be difficult to decide how to recover from the attack. In this scenario, both the electronic tallies and the paper records are untrustworthy, so in the worst case the only recourse may be to hold another election.

- If the bugs are fixed:

  > In this case, we do not see any realistic threat to the integrity of the VVPAT records.

In principle, if a malicious individual is able to introduce a malicious AccuBasic script, one might imagine a possible attack vector where the AccuBasic code prints false VVPAT records. However, in practice we do not see any viable threat here. AccuBasic scripts do have the capability to print to the AV-TSx printer, and this printer is shared for both printing reports (e.g., the zero tape, the summary report) during poll opening/closing, and for printing VVPAT records during the election. In theory, one might be able to envision a malicious AccuBasic script that, after it finishes printing the zero tape, continues running, waits some period of time, and then prints some text designed to look like a VVPAT record in hopes that this will be spooled into the security canister along with other VVPAT records. In practice, we believe that poll workers are unlikely to be fooled by this. As far as we can tell, the AV-TSx is single-threaded, so if the AccuBasic script does not relinquish control, the TSx will not show a startup screen welcoming voters to begin voting. It does not seem particularly likely that a poll worker would print and tear off a zero tape, feed the paper into the security canister, walk away before the machine has displayed a welcome screen, and fail to notice the machine printing and scrolling the tape into the security canister when there is no voter present. It is hard to imagine how this could be used for any kind of large-scale attack without being detected in at least some fraction of the polling places where the attack occurs.

Therefore, we consider this risk to be minimal, if the bugs in the AV-TSx AccuBasic interpreter are fixed.

Finding 2 *Everything we saw in the source code is consistent with Harri Hursti's attack on the AV-OS.*

Our analysis of the source code is consistent with Harri Hursti's findings that (a) the AccuBasic script on the AV-OS memory card can be replaced with a malicious script, (b) the vote counters on the AV-OS memory card can be tampered with and set to non-zero values, and (c) it is possible to use a malicious AccuBasic script to conceal this tampering by printing fraudulent zero reports or summary reports. Our source analysis confirmed that a malicious AccuBasic script is able to print to the printer (on both the AV-OS and the AV-TSx), display messages on the LCD display (on the AV-OS), and prompt for user responses (on the AV-OS). Our analysis also confirmed that the AV-OS fails to check that the vote counters are zero at the start of election day. We also confirmed that the AV-OS source code has numerous places where it manipulates vote counters as 16-bit values without first checking them for overflow, so that if more than 65535 votes are cast, then the vote counters will wrap around and start counting up from 0 again. (It is a feature of 16-bit unsigned computer arithmetic that large positive numbers just less than 65536 are effectively the same as small negative numbers)[6]. There is little doubt in our minds that Hursti's findings about the AV-OS are accurate. Even if the bugs we found in the AccuBasic interpreter are fixed, Hursti's attacks will remain possible.

[6]We discovered that the code does contain a check to ensure that it will not accept more than 65535 ballots. On the surface, that might appear adequate to rule out the possibility of arithmetic overflow. However, as Hursti's attack demonstrates, the existing check is not, in fact, adequate: if the vote counter started out at some non-zero value, then it is possible for the counter to wrap around after counting only a few ballots. This is a good example of the need for defensive programming. If code had been written to check for wrap-around immediately before every arithmetic operation on any vote counter, Hursti's technique of loading the vote counter with a large number just less than 65536 would not have worked.

The AV-TSx also appears to be at risk for similar attacks. The AV-TSx memory card also contains an AccuBasic script and appears to be vulnerable to similar kinds of tampering, unless the cryptographic keys have been updated from their default values (see below for a discussion).

Finding 3 *The AV-TSx (but not the AV-OS) contains cryptography designed to protect the contents of the AV-TSx memory card from modification while it is in transit. This mechanism appears to be an acceptable way to protect AccuBasic scripts from tampering while the memory card is in transit, assuming election officials update the cryptographic keys on every AV-TSx machine.*

The AV-TSx uses a cryptographic message authentication code (MAC), which ensures that it is infeasible for anyone who does not know the secret cryptographic key to tamper with the data stored on the memory card. The use of the cryptographic MAC in the AV-TSx appears to be an acceptable way to protect AccuBasic object code (.abo files) from tampering while the memory card is in transit, provided that election officials update the cryptographic keys on every AV-TSx. On the other hand, if the cryptographic keys are not updated, then the cryptographic mechanism does not protect against tampering with the contents of the memory card, for the following reasons.

The AV-TSx contains a default set of cryptographic keys. There is a procedure that election officials can use to change the keys stored on any particular AV-TSx machine. However, if this procedure is not performed on an AV-TSx machine, then that AV-TSx continues to use its default keys.

The default keys provide no security. They appear to be the same for all TSx machines in the nation, and in fact were discovered and published two and a half years ago (see Finding 4 below). Unfortunately, if no special steps are taken, the AV-TSx silently uses these insecure keys, without providing any warning of the dangers. Therefore, election officials will need to choose a new key for the county and update the keys on every AV-TSx machine themselves. Fortunately, there is a process for updating the keys on the AV-TSx by inserting a special smartcard into the AV-TSx machine.

So long as this process is followed, the cryptographic message authentication code (MAC) should provide acceptable security against tampering[7]. Because the AccuBasic script (.abo file) is stored on the memory card, the cryptography protects the AccuBasic script from being modified. If the cryptographic keys have been set properly, this defends against attacks like Harri Hursti's against the TSx: it prevents a malicious individual from *successfully* tampering with the AccuBasic script or the ballots stored on the memory card, even if the individual has somehow gained unsupervised access to the memory card, because the cryptographic check built in to the TSx firmware will fail and the TSx will print a warning message and refuse to proceed further.

The cryptographic MAC on the TSx appears to cover almost everything stored on the memory card data file. It covers election parameters, vote counters, the AccuBasic script (.abo file), and some other configuration data. The only exceptions we are aware of is that the file version number and the election serial number do not appear to be covered by the cryptographic MAC or by any checksum. These exceptions seem to be harmless.

In effect, the cryptography acts as the electronic equivalent of a tamper-resistant seal. If the contents of the memory card is tampered with, the cryptography will reveal this fact.

[7]We assume that the cryptographic keys are not stored on the memory card, but are stored on non-removable storage. We were not able to verify this assumption from the source code alone, but we have no reason to believe otherwise.

We stress that, like a tamper-resistant seal, the cryptography *only* defends against tampering while the memory card is in transit. The cryptography does *not* protect against tampering with AccuBasic scripts while they are stored on the GEMS server. In the Diebold system, the cryptographic protection is applied by the GEMS server when the memory card is initialized. The GEMS server stores the cryptographic keys and uses them to compute the cryptographic MAC when initializing a memory card; later, the AV-TSx uses its own copy of the keys to check the validity of the MAC. Of course, anyone who knows the cryptographic key can change the contents of the card and re-compute the MAC appropriately. This means that anyone with access to the GEMS server will have all the information needed to make undetected changes to AV-TSx memory cards. Also, AccuBasic scripts (.abo files) are stored on the GEMS server and downloaded onto memory cards as needed. If the copy of the .abo files stored on the GEMS server were corrupted or replaced, then this could affect every AV-OS machine and every AV-TSx machine in the county. In other words, if the operator of the GEMS server is malicious, or if any untrusted individual gains access to the GEMS server, all of the machines in the county could be compromised. The AV-TSx cryptography provides no defense against this threat; instead, it must be prevented by carefully guarding access to the GEMS server.

The cryptographic algorithm used in the AV-TSx, while perhaps not ideal, appears to be adequate for its purpose. The AV-TSx uses the following MAC algorithm:

$$F_k(x) = \mathrm{AES}_k(\mathrm{MD5}(x)),$$

where $\mathrm{AES}_k(\cdot)$ denotes AES-ECB encryption of a 128-bit value under key k. This choice of MAC algorithm is probably not what any cryptographer would select today, but it appears to be adequate. In August 2004, cryptographers discovered a way to find collisions in MD5, which prompted many cryptographers to suggest using some other hash algorithm in new systems. Fortunately, these collision attacks do not appear to endanger the way that AV-TSx uses its MAC, because chosen-plaintext attacks do not appear to pose a realistic threat. In contrast, the discovery of second pre-image attacks on MD5 would probably suffice to break the AV-TSx MAC algorithm, but fortunately no practical second pre-image attacks on MD5 are known. Consequently, given our current knowledge, the AV-TSx MAC appears to be acceptable.

In the long run, it would probably make sense to migrate to a more robust MAC algorithm (e.g., AES-CMAC). Even better, a cryptographic public-key signature (e.g., RSA, DSA) would appear to be ideal for this task. With the current scheme, anyone who can gain access to and reverse-engineer an AV-TSx machine can recover the cryptographic key and attack the other memory cards in the same county; while a public-key signature would eliminate this risk. Nonetheless, for present purposes the current scheme appears to be strong enough that it is not the weakest point in the system.

Finding 4 *The AV-TSx contains default cryptographic keys that are hard-coded into the source code and that are the same for every AV-TSx machine in the nation. One of these keys was disclosed publicly in July, 2003, yet it remains present in the source code to this day.*

We mentioned above that the AV-TSx contains a set of default keys that are used if the cryptographic keys have not been explicitly updated. We found that these default keys are hard-coded in the source code and are the same for every AV-TSx machine in the nation.

The presence of hard-coded keys in the TS was first disclosed in a famous scientific paper by Kohno, Stubblefield, Rubin, and Wallach in July, 2003. Their paper also revealed the value of the key—namely, `F2654hD4`—to the public. Subsequent reports from Doug Jones revealed that this design defect dates back to November, 1997, when he discovered the same hard-coded key and reported its presence to the vendor. These authors pointed out that use of a hard-coded key that is the same for all machines is very poor practice and opens up serious risks. It would be like a bank using the same PIN code for every ATM card they issued; if this PIN code ever became known, the exposure could be tremendous. It had been our understanding that all of the vulnerabilities found in those investigations two years ago had been addressed. It is hard to imagine any justification for continuing to use this key after it had been compromised and revealed to the public. This is a serious lapse that we find hard to understand considering how widely publicized this vulnerability was.

This also illustrates the reason that cryptographers uniformly recommend against hard-coded keys. If those keys are ever compromised or leaked, the compromise can affect every machine ever manufactured, and it can be difficult to change the key on every affected machine.

The AV-TSx would be more secure if it were changed to avoid use of default keys, i.e. if election officials were *required* to generate and load a county-specific cryptographic key onto the AV-TSx before its first use, and if the AV-TSx were to refuse to enter election mode if no key has ever been loaded.

Finding 5 *The AV-OS stores the four-digit supervisor PIN on the memory card. The PIN is stored in an obfuscated format, but this obfuscation offers limited protection due to its reliance on hard-coded magic constants in the source code.*

On the AV-OS, the four-digit PIN is derived as a specific function of a field stored on the memory card and of some constant values that are hard-coded into the source code. These magic constants are the same for every AV-OS machine across the nation, which is the rough equivalent of the hard-coded keys found in the AV-TSx. Thus, the AV-OS contains a design defect that is roughly similar to one in the AV-TSx.

Anyone with access to the AV-OS source code can learn these magic constants. Likewise, anyone who has unsupervised access to an AV-OS machine and the ability to perform reverse engineering could learn these magic constants. Once the magic constants are known, anyone who gains access to a memory card can read its contents and predict its four-digit PIN. Likewise, if they had unsupervised access to the memory card, they could set the four-digit PIN to any desired value by setting the field stored on the memory card appropriately. The use of the same magic constant values for every AV-OS machine in existence poses the risk that, if these constant values are ever disclosed, the security of the PIN protection would be undermined.

At present, we believe the security risks of this design misfeature are probably minor and limited in extent, because even knowledge of the PIN only provides a limited degree of additional access. There are worse things that an individual could do if she gained unsupervised access to an AV-OS memory card. Nonetheless, we caution election administrators not to place too much reliance on the four-digit PIN on the AV-OS.

Finding 6 *The AccuBasic interpreter was fairly cleanly structured and was organized in a way that made the source code very easy to read.*

The source code for the AccuBasic interpreter was written in a way that made it easy for us to understand its intent and operation and analyze its security properties. The code was split into many small functions whose purpose was clear and that performed one simple operation. There were comments explaining the purpose of each function and explaining tricky parts of the code. The clarity of the interpreter source code was about as good as any commercial code we have ever reviewed.

The interpreter is structured as a recursive descent parser, so that the program's call stack mirrors the stack of the associated context-free automaton. In addition, there is a global variable holding the global interpreter context: e.g., AccuBasic registers, AccuBasic variables, and various loop indexes. This was a reasonably elegant way to structure the implementation.

There were some ways that the implementation could have been improved. The code didn't use defensive programming, which would have helped tremendously to harden it against many malicious attacks. Also, the source code didn't document the relevant program invariants and pre-/post-conditions. We were forced to work these out by hand (e.g., that certain parameters were never `NULL`, that the global string register would never contain a string more than 255 bytes long, and so on), and it would have helped if these had been documented in the source code. Nonetheless, on the whole the interpreter source code was structured in a way that simplified the source code review task.

Finding 7 *The AccuBasic language is not a general-purpose system; it is narrowly tailored for its purpose.*

The AccuBasic language is *not* a full, general-purpose scripting language in the same category as, say, Visual Basic, in spite of the similarity of names. Instead, it is very modest in scope, with strongly circumscribed capabilities. If you are going to use an interpreted language at all in a context where security is important, this is the right way to do: one should include only the absolute minimum functionality in the language necessary to do the job it is designed for, and AccuBasic seems to meet that goal. In particular, we note that:

- AccuBasic is computationally complete in the sense that it can *compute* anything, but its interactions with the rest of the codebase are very limited. The parts of the firmware and operating system that it can invoke makes it basically useful *only* for printing reports, which is the intent.

- The AccuBasic interpreter cannot invoke most of the functions available in the firmware. It cannot read or write memory outside the its own stack. It can only invoke a handful of benign services necessary for its report-writing function, e.g. reading (but not writing) the vote totals or ballot file, accepting yes/no input from the user, writing to the printer, LCD screen, or touchscreen, appending an event to the audit log file, and reading the date and time.

- In particular, the AccuBasic interpreter has only read-only access to the vote counters or ballot file, so that AccuBasic scripts can construct reports from them, but cannot modify them.

In the short, the design of the AccuBasic language appears to us to be appropriate for its purpose.

Finding 8 *The AccuBasic interpreter cannot be invoked while the AV-OS or AV-TSx are executing the core election functionality, i.e., while they are accepting votes during the middle of election day.*

The AV-OS. We determined the AV-OS does not invoke the interpreter during the tallying of live election ballots. The AV-OS invokes the interpreter during pre-election procedures, such as printing test ballot zero reports and tallies, printing election zero reports, and printing labels for duplicate memory cards and audit reports. The AV-OS also invokes the interpreter to print post-election reports after the "ender" card is read.

The AV-TSx. We determined the AV-TSx does not invoke the interpreter while it is in "election" mode. The AV-TSx can invoke the interpreter under five circumstances:

1. Printing a zero report on machine initialization.
2. The "Print Election Results" button on the pre-election menu page for printing pre-election test results.
3. Printing election totals after a poll worker presses the "End Voting" button on the election menu page.
4. The "Print Election Results" button on the post-election menu page.
5. The "Print Results" button on the the accumulator menu page.

None of these can occur during the middle of the day while the TSx is in the process of interacting with voters and accepting votes.

These observations are also positive design points. The interpreter is not only very limited in its functionality, but it is very limited in the window of time during an election that it runs, which is what one wants when security is important.

Finding 9 *The AccuBasic interpreter does not appear to have been written using high-assurance software development methodologies.*

The AccuBasic interpreter appeared to be written using commercial standards of software development. This means it is not high-assurance software, nor was it developed following high-assurance methodologies.

High-assurance methods are often used for software systems where security is of utmost importance, most notably for military applications (e.g., software used to process classified documents). At a high level, these methods are similar to those used to build safety-critical software systems, where failure of the software can lead to loss of life (e.g., software found in avionics control systems, nuclear reactors, manned space flight, train control systems, automotive braking systems, and other similar settings).

In high assurance software development, one first determines explicitly what requirements the software and/or system must meet. One then designs the system, demonstrating throughout that the design meets the requirements. The method used to demonstrate this depends upon the degree of assurance desired. One then implements the system, and again justifies that the implementation meets the design. Indeed, one should be able to point to each requirement and show *exactly* what code is present as a result of that requirement. Finally, the operating instructions and procedures for the system and software must also meet the requirements.

We saw no evidence that the AccuBasic interpreter was developed in this way. Indeed, the problems we found argue against it. We should note that we did not see *anything* beyond the code—no

Informal name	Code identifier	Summary
“uninitialized”	`STAT_UNUSED`	`/* Empty formatted memory card. */`
“downloaded”	`STAT_DOWNLOADED`	`/* Downloaded memory card - pre-election mode. */`
“election mode”	`STAT_ELECTION`	`/* Election counting mode. */`
(none)	`STAT_ELECTION_DONE`	`/* Ender card fed, printing totals report. */`
(none)	`STAT_DONE`	`/* Post-election mode - ready for upload. */`
(none)	`STAT_UPLOADED`	`/* Upload done, ready for audit. */`
(none)	`STAT_AUDIT_DONE`	`/* Final audit report printed. */`

Figure 1: The modes that the AV-OS memory card can be in. For each mode, we list the informal name we use in this report, the symbolic name found in the source code, and a brief description taken from comments in the source code.

requirements documents, architecture documents, design documents, threat model documentation, or security analysis documents—all of which would be present were high assurance development techniques used.

We also expect that if one were going to use high-assurance programming practices anywhere in a voting system, the interpreter would be one of the most likely places to use it. If high-assurance practices had been used during the design and implementation of the AV-OS and AV-TSx, the vulnerabilities we found would likely have been avoided.

Finding 10 *The AV-OS is at risk from Harri Hursti's attacks no matter what state the memory cards are in when they are transported to the polls. Even if the memory cards are not put into election mode until the polls are opened, Hursti's attack is still possible.*

The AV-OS can be in one of several modes (e.g., pre-election, election mode, post-election). This is determined by a value stored on the memory card. It has been suggested that, if election workers were to wait to put the card into election mode until polls are opened, this might provide some level of defense against Hursti's attack. We find that this scheme does not, in fact, provide any useful protection.

Because the mode is stored on the memory card, whether or not the memory card is in election mode while in transit makes essentially no security difference. An attacker who can modify the object code and vote counts on the memory card (as Mr. Hursti did) could just as easily modify the election mode indicator too. In addition, all of the vulnerabilities described earlier (due to bugs in the code) are still exploitable, no matter what mode the memory card is in.

A detailed technical analysis of the election mode issue can be found in Section 4.1.

4.1 Technical details: Election mode and the AV-OS

In the AV-OS, memory cards can be in one of 7 modes, indicated by a field stored on the memory card (namely, `mCardHeader.electionStatus` in the source code). The states are documented in Figure 1. The mode of the memory card at the time when the machine is booted determines what functions the AV-OS will execute. The AV-OS also updates the mode of the card in response to operator input.

The memory card also contains many counters, including candidate counters (which contain, for each candidate, the number of votes cast for that candidate), race counters (which contain, for each race, the number of votes cast in that race), and card counters (which contain the total number of "cards cast" or, in other words, the number of ballots scanned). In each case, there are three values stored: the number of absentee votes, the number of election-day votes, and the total number of votes (which should be the sum of the previous two values). This reflects the fact that the machine can be set into a mode to count absentee votes or to count at the polling place. Note that there is some redundancy among these counter values: for instance, under normal operation, if Smith and Jones are the only two candidates in one race, then the race counter should equal the sum of Smith's candidate counter and Jones' candidate counter.

In Harri Hursti's demonstration, apparently the memory card was already placed into "election mode" before Hursti was given the card. It has been suggested that if the card had been in one of the two pre-election modes ("initialized" or "downloaded") when it was given to Hursti, then the Hursti attack would not work, because the process of placing the card into "election mode" would cause the vote counters to be zeroed.

Recall that Hursti's attack, in its most dangerous form, involved two components: (a) modifying the vote counters on the memory card to pre-load it with some non-zero number of votes for each of the candidates (e.g., +7 votes for Smith and −7 votes for Jones); (b) replacing the AccuBasic script with a malicious script that falsely printed a zero report showing zeros, even though the vote counters were in fact not zero. The ability to print a false zero report enabled Hursti to conceal the fact that he had stuffed the digital ballot box. This attack was demonstrated in a scenario where the card was set into "election mode" in the warehouse, before there was an opportunity to tamper with its contents. Might it perhaps be possible to defeat this attack if memory cards were left in pre-election mode at the warehouse, transported in this mode, and then poll workers were asked to set the card to "election mode" at the opening of polls? The idea is that, in the process of setting the card into "election mode", the AV-OS will zero out the vote counters on the card, thereby undoing any pre-loading of the memory card with fraudulent votes that might have occurred before that point. We were asked to characterize the behavior of election mode and investigate whether defenses of this form would provide any value in defending against Hursti's ballot stuffing attack.

Boot behavior. When starting the AV-OS machine, the operator has the option of holding the YES button or the YES and NO buttons (simultaneously) to execute special diagnostic, supervisory, and setup functions. When the machine boots, it will enter one of several modes, depending on how it is started up:

- If the operator holds the YES and NO buttons down while machine is booting, the machine enters diagnostics mode. In diagnostics mode, the operator can set the clock, dump the memory card image via a serial port, and test various physical components of the voting machine.
- If the operator holds only the YES button and the card is initialized (i.e., in any state other than "initialized", or in other words, `mCardHeader.electionStatus` $\neq$ `STAT_UNUSED`)), then it gives the operator the option to enter supervisor mode. To enter supervisor mode, the operator must enter the four digit PIN. In supervisor mode, the operator can modify the setup parameters, duplicate or clear the memory card, re-enter election mode after an "ender" card has been read, and reset the card to pre-election mode. In setup mode, the operator can change the phone number and configure the autofeeder and other physical devices.
- If the card is "uninitialized" (`mCardHeader.electionStatus` = `STAT_UNUSED`), the machine enters the aforementioned setup mode. Curiously, in this case the operator can enter setup

mode without entering a PIN. This means that it would be possible in this case to change the phone number it dials to transmit election results, without entering a PIN. (We are not aware of any California jurisdiction that uses the AV-OS's modem capabilities, so this is of little practical relevance in California.)

After these functions complete or if the operator chose not enter them, the machine displays

```
SYSTEM  TEST
***  PASSED  ***
```

and enters the main control loop. The main control loop works as follows:

- If the card state is "initialized" (`STAT_UNUSED`) or "downloaded" (`STAT_DOWNLOADED`), the machine executes pre-election functionality. Then, the machine goes back to the beginning of the loop.
- If the card state is in "election mode" (`STAT_ELECTION`), the machine executes the election functionality and begins accepting and counting ballots. Then, the machine goes back to the beginning of the loop.
- If the card state is in any of the four post-election states (`STAT_ELECTION_DONE`, `STAT_DONE`, `STAT_UPLOADED`, or `STAT_AUDIT_DONE`), it executes the post-election functionality. Then, the machine goes back to the beginning of the loop.

The behavior of the AV-OS. We focus on three modes, "uninitialized", "downloaded", and "election mode", and describe how the AV-OS behaves when loaded with a card in one of those three states.

If the card is "uninitialized", the AV-OS enters a mode of operation for downloading data to the memory card. If the download is successful, the operator can print an optional zero report using the AccuBasic interpreter and then the card is set to "downloaded" mode. At this point, or if a card in "downloaded" state is inserted into the AV-OS at any time, the AV-OS provides the operator with the option of performing pre-election testing. Pre-election testing includes reading blank and full marked ballots, counting test ballots, moving the ballot deflector, testing upload of results, and printing test total and audit reports.

After testing, the machine prompts the operator if he or she wants to enter election mode. If the operator answers yes, then the card is set to "election mode" (i.e., the field `mCardHeader.electionStatus` on the card is set to the value `STAT_ELECTION`) and the AV-OS proceeds to clear the election counters. The step of entering election mode zeroes out the card counters, race counters, and candidate counters. In other words, it clears the number of votes registered for each candidate, the number of votes registered in each race, and the total number of "cards cast" (i.e., the number of ballots scanned).

After the counters are zeroed, the AV-OS machine begins executing election functionality. This code first checks the card for errors. Then, it checks if any ballots have yet been counted by checking a counter stored on the memory card containing the total number of ballots that have been counted (`mCardHeader.numBalCounted[CTR_TOTAL]`). If no ballots have been counted, the AV-OS invokes the AccuBasic interpreter to print a zero report (without first prompting the operator) and then begins to accept and count ballots. If this counter is non-zero, then it skips the zero report step and immediately begins to accept and count ballots.

The proposed defense. The Hursti attack works by maliciously preloading some of the vote counters with fraudulent non-zero values. It was suggested to us that having poll workers putting the card into election mode at the polling place would defeat this attack, but it wasn't clear whether this would involve delivering memory cards in the "uninitialized" or "downloaded" state.

We believe that transporting memory cards to the polling place in the "uninitialized" state doesn't make much sense. This would mean that the cards have not been programmed and initialized yet. It seems unlikely poll workers would be expected to program and initialize the memory cards.

Therefore, we assume that this procedural defense would involve initializing memory cards at the county headquarters, so that when they arrive at the polling place they are in the "downloaded" state. This means that the memory cards will have been programmed and initialized and are ready to be put into election mode when the AV-OS machine is turned on. After the machine starts and completes the optional diagnostics mode (see above), it will prompt the operator (in order) to:

1. To count test ballots (optional);
2. To move the ballot deflector (optional);
3. To test the upload option (optional);
4. To print a totals report (optional);
5. To print an audit report (optional);
6. To prepare for the election (optional);
7. To enter supervisor mode (optional).

To enter election mode, the operator should answer yes to the 6th prompt. At that time, the AV-OS machine will clear the counters (see above) and start counting ballots.

Analysis. Unfortunately, the proposed defense against Hursti's attack is not effective. An adversary with access to the memory card could maliciously set the card into election mode, by setting the `mCardHeader.electionStatus` field on the card to the value `STAT_ELECTION` using a hex editor or by other means. When this card is inserted into the AV-OS, the AV-OS will not clear the counters, because the card is already in election mode. (The counters are only cleared when a card in the "downloaded" state is inserted into the AV-OS and explicitly put into election mode by the operator.)

On first consideration, one might expect that this attack could be detected. After all, an observant operator might notice that he or she did not have to navigate the prompts to explicitly put the machine into election mode, and thereby may be able to deduce that the card must have already been in election mode. Unfortunately, we cannot count on this defense, because things are more complex than they may initially appear.

Recall that if the memory card is in election mode and if the counter for the total number of ballots scanned (`mCardHeader.numBalCounted[CTR_TOTAL]`) is zero, then the AV-OS will execute an AccuBasic script to print a zero report before accepting ballots. The operator is not prompted before the AccuBasic script begins running. Of course, if we assume that an adversary has unsupervised access to the memory card while it is in transport, the adversary could have replaced the

AccuBasic script on the memory card with a malicious script, and this malicious script will start running as soon as the machine is turned on. Moreover, recall that AccuBasic scripts have the power to issue prompts to the LCD display on the AV-OS. This means that an adversary could write a malicious script which simulates the prompts the operator is expecting to see, to provide the illusion that the card is not already in election mode. When the operator answers yes to the 6th prompt, the AccuBasic script can print a zero report and exit, and the machine will start counting ballots.

In this scenario, as far as the operator can see, the machine will behave exactly as it would if the card had started in "downloaded" mode and if the operator had put it into election mode, clearing the counters. Nonetheless, in reality nothing could be farther from the truth. In this scenario, the card has been tampered with to pre-load it with votes, to set it into election mode so that these vote counters won't be cleared, and the AccuBasic script on the card has been tampered with so that the operator won't notice anything unusual and the zero report will not show these pre-loaded votes.

This shows that it is possible for an adversary to tamper with the memory card in a way that cannot be detected by the operator and that bypasses the clearing of the vote counters. In other words, even if memory cards are not put into election mode until the opening of polls, the election will still be vulnerable to a variation on Harri Hursti's attack. Therefore, it is our conclusion that procedures based on putting the AV-OS into election mode at the start of the day cannot be counted upon to protect the AV-OS machine against the vulnerabilities Harri Hursti found.

4.2 Checksums

We were asked to investigate what checksums exist in the AV-OS and AV-TSx, what types they are, and what they cover. We discuss these issues next.

Background. A checksum detects *accidental* changes to data. It reduces a large amount of data down to a fixed size value. This provides a level of redundancy: if the data is changed, then the checksum almost always changes as well. Hence, the checksum may provide a way to detect the change to the data.

Note that checksums are used to detect accidental changes to data values, but they are not at all useful in detecting malicious change. An example of an accidental change is a faulty memory cell on the memory card. If it cannot properly store the value it is supposed to, the computed checksum of the data will not equal the stored checksum, and a problem will be detected. On the other hand, if an adversary changes the data as well as all copies of the checksum value, it will be impossible to notice that the data was modified.

The AV-OS uses 16-bit checksums: a checksum can take on one of 65536 different values. The AV-OS computes numerous checksums over the data structures stored on the memory card. These checksum values are stored on the card and are also available to AccuBasic scripts to be printed in reports. A properly implemented checksum would likely detect any accidental corruption of the election setup parameters. Alternatively, a checksum printed over a memory card's vote totals at the close of polls could be compared with the same value at the county election offices to detect changes to the vote totals.

What is covered by the AV-OS checksums. The AV-OS memory card contains quite a few checksums. We list them, and what they cover, below:

1. *Election checksum:* covers the password, and flags controlling machine.
2. *Precinct checksum:* covers a few fields describing the precinct: its number, check digit, number of voters, sequence number, and precinct ID string.
3. *Precinct-card checksum:* covers fields that tie the precinct to the card structures.
4. *Race checksum:* all fields governing the race.
5. *Race counters checksum:* covers the total number of votes for each race, write ins, over-votes, under-votes, and blank votes.
6. *Candidate checksum:* covers the candidate number and party number.
7. *Candidate counters checksum:* covers all fields in the candidate structure.
8. *Card checksum:* covers all fields in the card.
9. *Card counters checksum:* covers the precinct number, card number, number of over-votes, under-votes, and blank votes for each card-counter.
10. *Voting positions checksum:* covers all fields governing where the candidate structure is.
11. *Text checksum:* covers all text fields (election title, vote center, vote date, straight party options, address, district name, race titles, and candidate names).
12. *Audit log checksum:* not used.

In summary, only some of the election setup parameters are covered by the AV-OS checksum. For example, the voting type field in the precinct (which governs whether it is early, absentee, or precinct voting) is not covered by any checksum. Additionally, the audit log is not covered by any checksum. It is difficult to determine how modifications to the fields not covered by the checksums could cause adverse effects, though it is a source of minor concern. Ideally, these checksums would cover all of the election parameters.

The AV-OS checksum algorithms. There are many ways to generate a checksum. The AV-OS code uses two separate techniques to compute a checksum. In the first, the checksum value is simply the arithmetic sum of the data being computed. As an example, if the vote counts were as follows:

Smith:	100
Jones:	32
Roberts:	7

then the checksum would be 139. If the value for any counter changes without the corresponding checksum value changing, it would be easy to notice the discrepancy and investigate what happened. However, using addition as a checksum, while simple to compute, fails to catch many classes of errors. For example, if the vote totals for Smith and Jones were switched, the checksum would still be 139. There are other classes of changes for which addition is not ideal and will not detect changes.

The AV-OS computes checksums over textual data in a slightly different, but related, manner. The checksum depends on the value of each of the names as well as their position (first, second, or so on).

The AV-OS checksum does not detect malicious attacks. An adversary with the ability to read and write to the memory card can always engineer the checksum to match what the malicious data they place. However, relying on the checksum to guarantee that data didn't change due to a malicious individual is not possible.

Using the addition operator (+) as a checksum may catch certain classes of non-malicious changes. However, an attacker can easily produce two different memory cards which have the same checksums. This means the checksum should not be used to determine malicious tampering. The textual checksum is also vulnerable to similar attacks.

If there was a desire to use checksums to detect malicious tampering with the contents of memory cards, a different checksum algorithm would be needed. One possibility would be to compute and print a cryptographic hash of the contents of the entire memory card at the beginning and end of the day, so that election officials can verify that the contents of the memory card had not been changed during transport. A cryptographic hash function is related to a checksum but instead of 65536 outputs, has over 2^{160} possible values; furthermore, it is specially designed to protect against reordering and malicious tampering. Examples of cryptographic hash functions include SHA-1 or SHA-256. If this route were taken, the cryptographic hash function should be applied to the entire contents of the memory card, including all election parameters and the audit log. Another possibility would be to use cryptographic digital signatures, either a public-key signature as discussed later, or a symmetric-key MAC like the one used by the TSx (see below).

The TSx "checksum". The AccuVote TSx operates differently. It reads the election parameters from a file on the memory card. There is a symmetric-key message authentication code (MAC) that protects the data from tampering. This computation depends on a secret key, and the MAC is designed so that anyone who does not know the key will not be able to tamper with the data without being detected. Thus, as long as the key is secret and unpredictable, it will detect malicious third party tampering, as well as problems with the storage media. A cryptographic MAC has all the advantages of a conventional checksum, in that it can detect accidental changes or corruption of the data, plus it can also detect malicious tampering as well. Thus, a cryptographic MAC is much better than a checksum in every way, and we expect the TSx to be extremely effective at detecting accidental data corruption.

See Finding 3 for a discussion of what data is protected by the cryptographic MAC on the TSx.

Since the TSx systems can read the AV-OS memory cards, they also include compatibility support for the data on those cards. Of course, those cards are only protected by the AV-OS checksums discussed earlier and are thus subject to the same caveats regarding tampering.

5 Mitigating the Risks

We next discuss several possible steps that could be taken to mitigate or ameliorate the risks discussed in this report. We start by discussing the full set of mitigations that might be possible in the long run; then, we discuss some short-term mitigation options.

5.1 Long-term Mitigation Strategies

Mitigation 1 *Adopt procedures that eliminate the possibility of a single person tampering with the memory card at any time during the lifetime of a memory card.*

One approach to mitigating the risk of tampering with the memory cards is to adopt various standard handling procedures that prevent someone from tampering without the risk of detection. These procedural controls would need be maintained throughout the lifetime of the memory card. They would affect procedures for writing memory cards at county offices, for opening and closing the polls, and for transport and storage of memory cards. Training of precinct judges and precinct clerks would need to be augmented to stress the critical nature of these procedural controls. Among the possibilities are these:

- Adopt the principle that no one should ever alone with memory cards, i.e. there should always be two or more persons present (or none). This parallels the common requirement that no one should be alone with ballots (blank or voted).
- Use numbered, tamper-evident seals to protect memory cards when they are stored or when they are inserted in a voting machine. Keep records, and train poll workers to monitor those seals and their numbers and report anomalies. No one person should be entrusted with that task; all poll workers should sign off that the seals were intact.
- Permanently affix serial numbers to the memory cards and adopt written chain-of-custody procedures for transfer of custody from one pair of people to another, including poll workers.
- Train all personnel, including poll workers, that memory cards are ballot boxes and must be treated with the same degree of care and security.
- Whenever the procedures outlined are breached for some reason, take the memory card(s) in question out of service and zero them (in the presence of at least two people) before using them again.

It would help if memory cards were sealed inside the AV-OS at county headquarters, and AV-OS machines delivered to the polling place with the card already inserted and protected by tamper-evident seals. At the close of polls, it would help if poll workers did not break the seal, but rather returned the entire unit (with memory card still sealed inside) to county headquarters. This would reduce the opportunity for poll workers to tamper with memory cards.

When the AV-OS is used as a central-count machine (e.g., to count absentee votes), similar processes could be used to ensure that officials never insert a memory card into the AV-OS unless they are sure no one has had unsupervised access to the memory card. Because central-count machines reside in a controlled environment with physical security protections, and only a limited number of individuals have access to them, it should be much easier to apply very strong procedural controls to these machines.

Mitigation 2 *Revise the source code of the AccuBasic interpreter to fix these vulnerabilities, introduce the use of defensive programming practices, and use security practices that will eliminate the possibility of any other vulnerabilities of the sort we discovered here.*

We can break this mitigation down into several (closely related) steps:

- Fix the AV-OS AccuBasic interpreter to eliminate the bugs we found. Every one of the bugs we found should be fixed. Any other bugs of the same sort should also be fixed.

It is not enough merely to introduce narrow changes to patch the specific bugs we found. Those bugs were symptoms of more fundamental flaws in the programming practices used to build the interpreter. The only way to be sure that all the bugs have been eliminated is to fix the root cause. We explain next what would be involved in doing so.

- Revise the interpreter source code, line by line, to eliminate all trust in the contents of the memory card. One of the reasons that these vulnerabilities existed was because the programmer implicitly assumed that the memory card would not be tampered with, and that the AccuBasic object code (.abo file) on the memory card was produced by a legitimate AccuBasic compiler. The source code should be changed to eliminate all instances of this kind of trust. For instance, when reading an integer from the memory card, the interpreter should first check that it is within the expected range. When reading a string from the memory card, the interpreter should not blindly assume that the string is `'\0'`-terminated, but should check that this is true before relying on it. Thus, this would involve identifying every point in the code that reads data from the memory card (or any other untrusted source) and inserting appropriate input validation checks at that point.

  Likewise, every place where the code manipulates a vote counter, the code should check that the vote counter is (a) non-negative, and (b) arithmetic on it (e.g., incrementing a vote counter) does not wrap or overflow. If the code always checked that every vote counter were non-negative, and eliminated all possibility of arithmetic overflow or wrap-around modulo 65536, Hursti would not have been able to pre-load a negative number of votes for one candidate on the memory card. If the code had checked that all vote counters were zero at the start of the day, Hursti would not have been able to pre-load a positive number of votes for any candidate, either.

  In addition, it would be prudent to revise the source code of the interpreter to prevent infinite loops and infinite recursion. One way to do this would be to introduce a timeout of some sort, and check for timeout every time the AccuBasic script executes any kind of backward jump, call, or control transfer.

- Revise the interpreter, line by line, to incorporate defensive programming throughout the code. If the code had been written to follow defensive programming practices in a more disciplined way, these vulnerabilities could not have existed.

  Programming and driving a car are similar in that the programmer, like the driver, cannot control his or her environment; he or she can merely control how the software, or the car, reacts to that environment. Driving courses emphasize "defensive driving". Driving students learn to prepare for other drivers taking unexpected, and dangerous, actions. They understand that they cannot control other drivers, and that they must avoid accidents even if those accidents are not their fault.

  Similarly, programmers should develop software with the understanding that the environment is not trusted. Users may enter incorrect input; system hardware may fail; touch screens may be miscalibrated and so return nonsensical values to the program. Good programming style is to build software that either functions correctly in the face of such errors, or else reports the error and terminates gracefully. This style of defensive programming is called "robust programming".

As an example, a buffer overflow occurs when an input is larger than the memory allocated to hold that input. The excess input can change internal values, causing the software to malfunction and return incorrect results. In some cases, this allows a malicious user to breach security. Robust programming requires that *every* input be checked; were this style followed, buffer overflows would not occur because the program would check the length of the input, determine it was too long, and reject it.

More generally, defensive programming generally means that every module should apply these checks to data it receives from other modules, and should refrain from trusting other modules. Just as drivers are taught that they cannot control what other drivers may do, defensive programming teaches that programmers cannot control what other modules may do, and so should treat them as untrusted and ensure that other modules cannot compromise their own integrity.

Thus, defensive programming often involves disciplined use of various idioms that ensure the safety of the code. Before copying a string into the buffer, one inserts code to check that there is sufficient room for the string. Before dereferencing a pointer, one writes code to check that the pointer is not NULL. Before adding two integers, one checks that the addition will not overflow. Code is added to perform these checks, even when they seem unnecessary, because sometimes one's assumption that the check is not necessary turns out to be inaccurate.

Our review of the interpreter source code showed that the programmers could have applied this principle of robust programming more extensively to the code. Specifically, the code had shortcomings (detailed above) that would not occur when software is designed and written to be robust. Hence, when the bugs in the AccuBasic interpreter are fixed, it seems prudent to also revise the code to be robust in the face of erroneous, unexpected, and malicious input, and other failures such as hardware failure.

- After the source code is revised, it would make sense to commission an independent source code review to confirm whether all of the vulnerabilities have been eliminated and to assess whether the code has used structured programming practices that are adequate to have confidence that no other security vulnerabilities of this sort are likely to be present.

If the source code is not revised, anyone with unsupervised access to a memory card, or with access to the GEMS server, may be able to exploit the vulnerabilities we found to take control of voting machines and compromise the electronic tallies. Such an attack might be able to cause lasting effects that persist across elections, and it is not clear whether there would be any way to repair the resulting damage. If the source code is revised to fix the vulnerabilities we found, these attacks would not be possible.

Even if the interpreter source code is fixed, it would still be possible for an individual who can introduce a malicious AccuBasic script to cause fraudulent zero tapes and fraudulent summary reports to be printed. Depending on whether the arithmetic overflows are fixed, such an individual might also be able to pre-load a memory card with a positive or negative number of votes for some candidates.

Mitigation 3 *Protect AccuBasic object code from tampering and modification, either by (a) storing AccuBasic object code on non-removable storage and treating it like firmware, or by (b) protecting AccuBasic object code from modification through the use of strong cryptography (particularly public-key signatures).*

All of the vulnerabilities we uncovered were due to the fact that part of the code of the voting system (namely, the AccuBasic object code) was not adequately protected from modification. Thus, one effective mitigation would be to protect the code from modification, using one of two strategies:

(a) Protect AccuBasic object code in the same way that the rest of the firmware object code is protected, by placing the AccuBasic object code on physically secured non-removable storage. Normally, firmware is protected from modification by storing it on a non-removable storage device (e.g., EEPROM) that is not easily externally accessible and that is protected from casual tampering through some kind of physical security protection. AccuBasic object code could be stored in the same way. If this were done, it would eliminate an entire attack vector, because attackers would no longer have the opportunity to replace the AccuBasic object code with a malicious AccuBasic script.

    Of course, in this approach AccuBasic code would need to be protected with the same protections that are afforded to firmware code. If there is any way to update AccuBasic object code (or any other code), the update process must be strongly authenticated, and updates to the AccuBasic object code must be authenticated as securely as updates to the firmware. (By *authenticated*, we mean that there are procedural and technological controls which ensure that only authorized individuals can update the code, and only under appropriate circumstances.)

    We recognize that different jurisdictions may require different AccuBasic scripts. One way to handle this would be for each jurisdiction to update the firmware with the appropriate AccuBasic script. Another possibility would be for the vendor to store all the different versions of AccuBasic object files that might ever be needed on the firmware, and for the memory card to contain an index (e.g., numbered from 1 to n, where n is the number of different AccuBasic scripts stored in the firmware) identifying which of these .abo files is to be used. Depending on the circumstances, this index might need to be protected from modification.

(b) Alternatively: Use strong cryptography to protect the AccuBasic object code while it is stored on removable media. The appropriate protection would involve signing the AccuBasic object code with a cryptographically strong public-key signature scheme (e.g., RSA, DSA, or some other appropriate public-key algorithm) and arranging for the firmware to check the validity of this signature before executing the AccuBasic code. The private key would need to be guarded zealously (e.g., using a hardware security module (HSM)). In addition, considerable thought needs to be given to key management as well as to which part of the data is signed by which principals (e.g., by the vendor, by the GEMS server, or by other authorities).

    While the AV-TSx cryptography is a good first step in this direction, it falls short in several respects:

    - The use of symmetric-key cryptography in the AV-TSx increases the risk of key exposure. It would be safer to use public-key (asymmetric) digital signatures for this purpose.
    - The use of hard-coded symmetric keys that are the same for all AV-TSx units is highly inappropriate for this purpose, and should be avoided at all costs.
    - The existence of any kind of default key is a usability pitfall, because it makes it possible for election officials to forget to change the keys, thereby leaving them unaware of their vulnerability. This is an additional problem with hard-coded symmetric keys. We recommend that default keys be avoided.

- Insufficient thought has been given to the topic of key management and which entities are in possession of the appropriate cryptographic keys.

  Fixing these shortcomings would prevent unauthorized individuals from introducing malicious AccuBasic scripts.

Of course, in both approaches the AccuBasic scripts need to be considered part of the codebase of the system, and should be reviewed as part of the qualification and certification process.

In the long run, the consequences of not protecting AccuBasic code from modification are that anyone who gains unsupervised access to memory cards can tamper with their contents, attack the voting systems (e.g., using Hursti-style attacks), and potentially manipulate the electronic vote tallies.

Mitigation 4 *Change the architecture of the AV-OS and the AV-TSx so they do not store code on removable memory cards.*

In the long run there are good reasons for changing the AV-OS and AV-TSx software architectures so that they do not rely on interpreted code stored on a removable memory card, or that they do not use interpreted code at all and eliminate AccuBasic. All of the potential vulnerabilities discussed here are rooted in the fact the code is stored on the removable memory cards, and these cards are handled by, and in the custody of, many people in a major election. There does not seem to be any *fundamental* reason why the AccuBasic code cannot be part of the firmware codebase, rather than stored on the removable memory card. That change would not only eliminate these attacks, but some GEMS-based attacks on the code as well. Of course there would need to be enough firmware storage space in the machines to hold the AccuBasic code, but we suspect that is not an insoluble problem. This change would reduce the vendor's flexibility in providing different reporting options to different jurisdictions (i.e. different AccuBasic scripts). But if it is accepted that the AccuBasic scripts are part of the voting system "code", as they are, and that therefore they must be subject to testing and code review by federal and state examiners, then that flexibility would be lost anyway, since it cannot be expected that the examiners would be able to study hundreds of variations on the AccuBasic script packages produced for different jurisdictions.

Mitigation 5 *Change the architecture of the AV-OS and the AV-TSx so they do not contain any interpreter or use any kind of interpreted code.*

There are also good arguments for eliminating AccuBasic interpreted code entirely from voting system software. The FEC 2002 Voluntary Voting System Standards expressly forbid interpreted code in section 4.2.2. Perhaps the standard writers had in mind forbidding only powerful, interpreted *programming* languages, such as Visual Basic, and not relatively benign and limited *rendering* languages such as HTML. AccuBasic falls somewhere in the middle on the more benign side (assuming the interpreter bugs are fixed). But the text of the standard is pretty clear, and the same language from the 2002 standards has been preserved in the EAC's new successor standard, the Voluntary Voting Systems Guidelines, as section 5.2.2. To be in compliance it would seem that AccuBasic would have to be eliminated, or the standard would have to be changed.

In any case, the inclusion of interpreted languages in a voting system causes great burdens on examiners and code reviewers, who have to be highly skilled and do considerable analysis of the compiler and interpreter in order to verify that it does not present security vulnerabilities or permit malicious code to go unnoticed. It seems untenable to us that every time there is a change to the AccuBasic language or interpreter another round of detailed code review such as we have done would be required; however, an interpreter is such a delicate and powerful feature (from a security point of view) that we cannot recommend shortcuts in its examination either.

5.2 Short-term Mitigation Strategies for Local Elections

One disadvantage of several of these mitigation strategies (e.g., revising or eliminating the AccuBasic interpreter, improving the cryptography, etc.) is that changes to the source code will incur significant delays. Source code changes would need to be approved by the federal qualification process as well as the state certification process. Therefore, in the short term it seems appropriate to consider mitigation strategies that do not involve changing the source code.

For local elections (i.e., elections that do not span the entire state), we believe there are mitigation strategies that could be viable for the short term. For instance, one possibility might be the following two-prong approach:

- For the AV-TSx, update the cryptographic keys on every AV-TSx machine and rely on the cryptography to prevent tampering with memory cards. Election officials would need to first choose a secret and unguessable cryptographic key. The new cryptographic key should be chosen at random by county staff, should not be divulged to anyone, not even the vendor (because anyone who learns the secret key gains the ability to tamper undetectably with memory cards), should not be shared across counties, and should be tightly controlled. Then, the process of updating the keys requires inserting a smartcard into every AV-TSx machine. Officials could adopt checklists or some other process to ensure that every AV-TSx machine has had its keys updated before it is sent into the field. Election officials should be warned that if they forget to change the cryptographic keys, the machine will outwardly appear to function correctly, but will be vulnerable to attack.

- For the AV-OS, deploy strict procedural safeguards to prevent anyone from gaining unsupervised access to a memory card. We would suggest dual-person controls over the entire lifecycle of the memory card, chain of custody provisions, and use of numbered tamper-evident seals. It would also help to load and seal the memory card into the AV-OS unit at the warehouse in advance of the election, ship it in this state, and when the election is over, have poll workers return the entire machine (with the memory card still sealed inside) to the county collection point, where election officials would check that the seal remains undisturbed and record the number on the seal before removing the memory card. This would ensure that the memory card is protected by a tamper-evident seal for the entire time that it is outside the control of county staff and would reduce the opportunities for someone to tamper with the memory card while it is in transit. We recognize that these heightened procedural protections are likely to be somewhat burdensome, but as a short-term protection (until the source code can be fixed), they may be appropriate. See Mitigation 1 for further discussion of procedural mitigations.

While these strategies do not completely eliminate all risk, we expect they would be capable of reducing the risk to a level that is manageable for local elections in the short term.

In the longer term, or for statewide elections, the risks of not fixing the vulnerabilities in the AccuBasic interpreter become more pronounced. Larger elections, such as a statewide election, provide a greater incentive to hack the election and heighten the stakes. Also, the longer these vulnerabilities are left unfixed, the more opportunity it gives potential attackers to learn how to exploit these vulnerabilities. For statewide elections, or looking farther into the future, it would be far preferable to fix the vulnerabilities discussed in this report.

6 Conclusions

We have detailed a number of security vulnerabilities in the AV-OS and AV-TSx implementations of the AccuBasic interpreter. In the long term, these vulnerabilities can be easily fixed and the risks eliminated or mitigated. We have made recommendations about several ways in which that might be accomplished. In the short term, we believe the risks can be mitigated through appropriate use procedures.

7 Glossary

.abo file a file containing AccuBasic object code (byte code)

AccuBasic a Diebold-proprietary programming language used (in slightly different versions) in both the AV-OS and AV-TSx machines; AccuBasic programs allow very limited control over the behavior of the voting system

buffer a fixed-size area of memory

buffer overrun a type of program bug in which the program attempts to write more data into a buffer than the buffers size permits. The extra data is thus written beyond the end of the buffer into other memory, where it often overwrites something else of significance, i.e. either other data, or control information, or even instructions. When that happens, the program is corrupted, and any of a vast number of unpredictable things might ensue. One common hacker attack is to deliberately take advantage of a buffer overrun bug, corrupting the program in a specific way that allows the hacker to do things he otherwise would not be able to do. (Usually the goal is to take complete control of the machine.)

byte code object code of a relatively simple kind (e.g., that happens to be encoded as characters (bytes) instead of binary data)

C a very widely used programming language

C++ another widely used programming language, more modern than C, and (roughly) including C as a subset

compiler a program that translates another program from its source language (the human readable form) into an object language (a form not so easily human readable, but much more convenient for machine execution). The AccuBasic compiler translates AccuBasic programs (source code) into AccuBasic object code (also known as byte code in this case).

file system hierarchical collection of files and directories (folders), along with their names, types, and the software to read and write them

firmware software resident inside the voting machine (i.e. not on a removable memory card) and that is (or should be) unmodifiable once the machine is in operation

hex editor an editor that can modify data directly at the binary level. (Hex refers to hexadecimal (base-16) arithmetic, which is extremely closely related to binary, but more compact.) A hex editor is a universal editor, in that it can edit absolutely any kind of digital data, although it requires some knowledge and skill to use it in any particular case.

interpreter a program whose function is to execute another program, usually one that is in the form of object code. The AccuBasic interpreter is part of the firmware of the AV-OS or AV-TSx, and executes AccuBasic object code, i.e. .abo files.

memory mapped memory mapped data is data that resides on some attached memory device, and yet is made to appear as if it is in main memory. (In the technical jargon, the data on the attached device is mapped onto a portion of the machines memory address space.)

object code a program represented in the form of discrete instructions that are easy for a computer (or an interpreter) to execute efficiently. It is more difficult for humans to read and write object code than source code, but it can be done with only modest skill. Usually object code is produced with the aid of a compiler, but it does not have to be.

scripting language a programming language designed primarily so that the programs written in it can easily manipulate character data and files (as opposed to, e.g. binary data), and can easily invoke and control other programs; AccuBasic can be described as a limited-purpose scripting language.

scripts programs written in a scripting language like AccuBasic

source code any software in the original form as written by a human programmer; this is the form in which code is easily read and written by programmers, but cannot be directly executed by a computer or an interpreter