This paper, copyright the IEEE, appears in *IEEE Symposium on Security and Privacy 2004*. IEEE Computer Society Press, May 2004. This paper previously appeared as Johns Hopkins University Information Security Institute Technical Report TR-2003-19, July 23, 2003.

Analysis of an Electronic Voting System

TADAYOSHI KOHNO[*] ADAM STUBBLEFIELD[†] AVIEL D. RUBIN[‡]
DAN S. WALLACH[§]

February 27, 2004

Abstract

With significant U.S. federal funds now available to replace outdated punch-card and mechanical voting systems, municipalities and states throughout the U.S. are adopting paperless electronic voting systems from a number of different vendors. We present a security analysis of the source code to one such machine used in a significant share of the market. Our analysis shows that this voting system is far below even the most minimal security standards applicable in other contexts. We identify several problems including unauthorized privilege escalation, incorrect use of cryptography, vulnerabilities to network threats, and poor software development processes. We show that voters, without any insider privileges, can cast unlimited votes without being detected by any mechanisms within the voting terminal software. Furthermore, we show that even the most serious of our outsider attacks could have been discovered and executed without access to the source code. In the face of such attacks, the usual worries about insider threats are not the only concerns; outsiders can do the damage. That said, we demonstrate that the insider threat is also quite considerable, showing that not only can an insider, such as a poll worker, modify the votes, but that insiders can also violate voter privacy and match votes with the voters who cast them. We conclude that this voting system is unsuitable for use in a general election. Any paperless electronic voting system might suffer similar flaws, despite any "certification" it could have otherwise received. We suggest that the best solutions are voting systems having a "voter-verifiable audit trail," where a computerized voting system might print a paper ballot that can be read and verified by the voter.

[*]Dept. of Computer Science and Engineering, University of California at San Diego, 9500 Gilman Drive, La Jolla, California 92093, USA. E-mail: tkohno@cs.ucsd.edu. URL: http://www-cse.ucsd.edu/users/tkohno. Most of this work was performed while visiting the Johns Hopkins University Information Security Institute. Supported by a National Defense Science and Engineering Graduate Fellowship.

[†]Information Security Institute, Johns Hopkins University, 3400 North Charles Street, Baltimore, Maryland 21218, USA. E-mail: astubble@cs.jhu.edu. URL: http://spar.isi.jhu.edu/~astubble.

[‡]Information Security Institute, Johns Hopkins University, 3400 North Charles Street, Baltimore, Maryland 21218, USA. E-mail: rubin@cs.jhu.edu. URL: http://www.avirubin.com.

[§]Dept. of Computer Science, Rice University, 3121 Duncan Hall, 6100 Main Street, Houston, Texas 77005, USA. E-mail: dwallach@cs.rice.edu. URL: http://www.cs.rice.edu/~dwallach.

Contents

1 Introduction **3**

2 System overview **5**

3 Smartcards **9**
- 3.1 Exploiting the lack of cryptography: Creating homebrew smartcards . . . 9
- 3.2 Casting multiple votes . . . 10
- 3.3 Accessing administrator and poll worker functionality . . . 10

4 Election configurations and election data **11**
- 4.1 Tampering with the system configuration . . . 12
- 4.2 Tampering with ballot definitions . . . 13
- 4.3 Impersonating legitimate voting terminals . . . 14
- 4.4 Key management and other cryptographic issues with the vote and audit records . . . 14
- 4.5 Tampering with election results and linking voters with their votes . . . 15
- 4.6 Audit logs . . . 17
- 4.7 Attacking the start of an election . . . 17

5 Software engineering **18**
- 5.1 Code legacy . . . 18
- 5.2 Coding style . . . 18
- 5.3 Coding process . . . 19
- 5.4 Code completeness and correctness . . . 20

6 Conclusions **21**

1 Introduction

Elections allow the populace to choose their representatives and express their preferences for how they will be governed. Naturally, the integrity of the election process is fundamental to the integrity of democracy itself. The election system must be sufficiently robust to withstand a variety of fraudulent behaviors and must be sufficiently transparent and comprehensible that voters and candidates can accept the results of an election. Unsurprisingly, history is littered with examples of elections being manipulated in order to influence their outcome.

The design of a "good" voting system, whether electronic or using traditional paper ballots or mechanical devices, must satisfy a number of sometimes competing criteria. The *anonymity* of a voter's ballot must be preserved, both to guarantee the voter's safety when voting against a malevolent candidate, and to guarantee that voters have no evidence that proves which candidates received their votes. The existence of such evidence would allow votes to be purchased by a candidate. The voting system must also be *tamper-resistant* to thwart a wide range of attacks, including ballot stuffing by voters and incorrect tallying by insiders. Another factor, as shown by the so-called "butterfly ballots" in the Florida 2000 presidential election, is the importance of *human factors*. A voting system must be comprehensible to and usable by the *entire* voting population, regardless of age, infirmity, or disability. Providing accessibility to such a diverse population is an important engineering problem and one where, if other security is done well, electronic voting could be a great improvement over current paper systems. Flaws in any of these aspects of a voting system, however, can lead to indecisive or incorrect election results.

ELECTRONIC VOTING SYSTEMS. There have been several studies on using computer technologies to improve elections [4, 5, 20, 21, 25]. These studies caution against the risks of moving too quickly to adopt electronic voting machines because of the software engineering challenges, insider threats, network vulnerabilities, and the challenges of auditing.

As a result of the Florida 2000 presidential election, the inadequacies of widely-used punch card voting systems have become well understood by the general population. Despite the opposition of computer scientists, this has led to increasingly widespread adoption of "direct recording electronic" (DRE) voting systems. DRE systems, generally speaking, completely eliminate paper ballots from the voting process. As with traditional elections, voters go to their home precinct and prove that they are allowed to vote there, perhaps by presenting an ID card, although some states allow voters to cast votes without any identification at all. After this, the voter is typically given a PIN, a smartcard, or some other token that allows them to approach a voting terminal, enter the token, and then vote for their candidates of choice. When the voter's selection is complete, DRE systems will typically present a summary of the voter's selections, giving them a final chance to make changes. Subsequent to this, the ballot is "cast" and the voter is free to leave.

The most fundamental problem with such a voting system is that the entire election hinges on the correctness, robustness, and security of the software within the voting terminal. Should that code have security-relevant flaws, they might be exploitable either by unscrupulous voters or by malicious insiders. Such insiders include election officials, the developers of the voting system, and the developers of the embedded operating system on which the voting system runs. If any party introduces flaws into the voting system software or takes advantage of pre-existing flaws, then the results of the election cannot be assured to accurately reflect the votes legally cast by the voters.

Although there has been cryptographic research on electronic voting [13], and there are new approaches such as [6], currently the most viable solution for securing electronic voting machines is to introduce a "voter-verifiable audit trail" [10, 20]. A DRE system with a printer attachment, or even a traditional optical scan system (e.g., one where a voter fills in a printed bubble next to their chosen candidates), will satisfy this requirement by having a piece of paper for voters to read and verify that their intent is correct reflected. This paper is stored in ballot boxes and is considered to be the primary record of a voter's intent. If, for

some reason, the printed paper has some kind of error, it is considered to be a "spoiled ballot" and can be mechanically destroyed, giving the voter the chance to vote again. As a result, the correctness of any voting software no longer matters; either a voting terminal prints correct ballots or it is taken out of service. If there is any discrepancy in the vote tally, the paper ballots will be available to be recounted, either mechanically or by hand. (A verifiable audit trail does not, by itself, address voter privacy concerns, ballot stuffing, or numerous other attacks on elections.)

"CERTIFIED" DRE SYSTEMS. Many government entities have adopted paperless DRE systems without appearing to have critically questioned the security claims made by the systems' vendors. Until recently, such systems have been dubiously "certified" for use without any public release of the analyses behind these certifications, much less any release of the source code that might allow independent third parties to perform their own analyses. Some vendors have claimed "security through obscurity" as a defense, despite the security community's universally held belief in the inadequacy of obscurity to provide meaningful protection [18].

Indeed, the CVS source code repository for Diebold's AccuVote-TS DRE voting system recently appeared on the Internet. This appearance, announced by Bev Harris and discussed in her book, *Black Box Voting* [14], gives us a unique opportunity to analyze a widely used, paperless DRE system and evaluate the manufacturer's security claims. Jones discusses the origins of this code in extensive detail [17]. Diebold's voting systems are in use in 37 states, and they are the second largest and the fastest growing vendor of electronic voting machines. We only inspected unencrypted source code, focusing on the `AVTSCE`, or AccuVote-TS version 4, tree in the CVS repository [9]. This tree has entries dating from October 2000 and culminates in an April 2002 snapshot of version 4.3.1 of the AccuVote-TS system. From the comments in the CVS logs, the AccuVote-TS version 4 tree is an import of an earlier AccuTouch-CE tree. We did not have source code to Diebold's GEMS back-end election management system.

SUMMARY OF RESULTS. We discovered significant and wide-reaching security vulnerabilities in the version of the AccuVote-TS voting terminal found in [9] (see Table 1). Most notably, voters can easily program their own smartcards to simulate the behavior of valid smartcards used in the election. With such homebrew cards, a voter can cast multiple ballots without leaving any trace. A voter can also perform actions that normally require administrative privileges, including viewing partial results and terminating the election early. Similar undesirable modifications could be made by malevolent poll workers (or janitorial staff) with access to the voting terminals before the start of an election. Furthermore, the protocols used when the voting terminals communicate with their home base, both to fetch election configuration information and to report final election results, do not use cryptographic techniques to authenticate either end of the connection nor do they check the integrity of the data in transit. Given that these voting terminals could potentially communicate over insecure phone lines or even wireless Internet connections, even unsophisticated attackers can perform untraceable "man-in-the-middle" attacks.

We considered both the specific ways that the code uses cryptographic techniques and the general software engineering quality of its construction. Neither provides us with any confidence of the system's correctness. Cryptography, when used at all, is used incorrectly. In many places where cryptography would seem obvious and necessary, none is used. More generally, we see no evidence of disciplined software engineering processes. Comments in the code and the revision change logs indicate the engineers were aware of some areas in the system that needed improvement, though these comments only address specific problems with the code and not with the design itself. We also saw no evidence of any change-control process that might restrict a developer's ability to insert arbitrary patches to the code. Absent such processes, a malevolent developer could easily make changes to the code that would create vulnerabilities to be later exploited on Election Day. We also note that the software is written entirely in C++. When programming in a language like C++, which is not type-safe, programmers must exercise tight discipline to prevent their programs from being vulnerable to buffer overflow attacks and other weaknesses. Indeed, buffer overflows

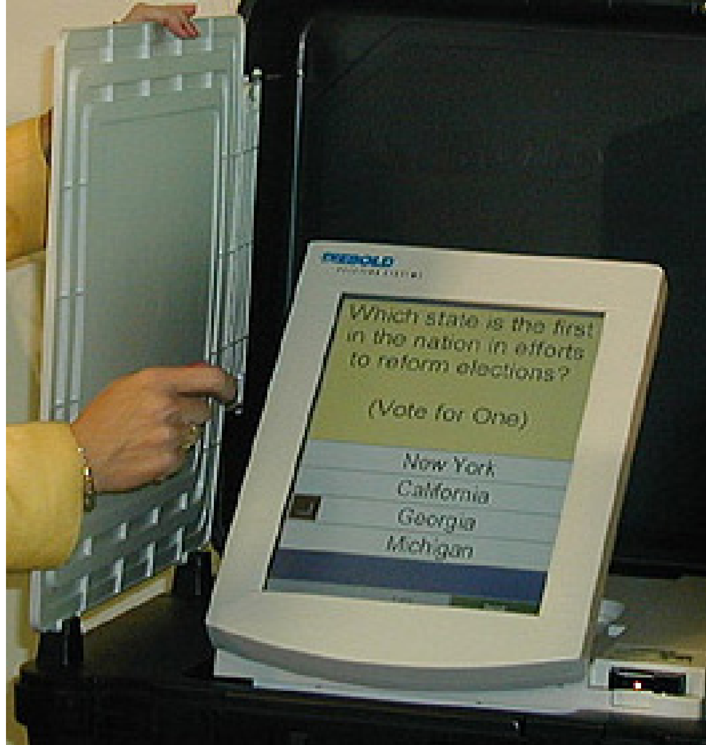

Figure 1: A Diebold AccuVote-TS voting machine (photo from `http://www.sos.state.ga.us/`). Note the smartcard reader in the lower-right hand corner.

caused real problems for AccuVote-TS systems in real elections.[1]

SUBSEQUENT WORK. Following the release of our results, the state of Maryland hired SAIC [27] and RABA [24] and the state of Ohio hired Compuware [7] to perform independent analyses of Diebold's AccuVote-TS systems. These analyses not only support our findings, but show that many of the issues we raise and attacks we identify still apply to recent versions of the AccuVote-TS system, and particularly to the machines recently purchased by Maryland. These analyses also identified security problems with the back-end GEMS server. Additionally, RABA's "red team" implemented some of our attacks in a mock election setting; e.g., they modified smartcards so that a voter could vote more than once (Section 3.2 and [24, page 16]) and they implemented our ballot reordering attack, thereby tricking voters to vote for the wrong candidates (Section 4.2 and [24, pages 18 and 21]). Jones discusses these three reports in more detail [17].

2 System overview

The Diebold AccuVote-TS 4.3.1 system we analyzed [9], which was written in C++, was designed to run on a Windows CE device, an example of which is shown in Figure 1. The code also compiles and runs (with slightly different configurations) on regular Microsoft Windows machines, thus enabling us to verify that the code represents a complete system. We shall refer to a device running the vote collection software as a *voting terminal*.

[1] `http://www.sccgov.org/scc/assets/docs/209815KeyboardAttachment-200440211.pdf` (page 60 of the report, page 61 of the PDF)

	Voter (with forged smartcard)	Poll Worker (with access to storage media)	Poll Worker (with access to network traffic)	Internet Provider (with access to network traffic)	OS Developer	Voting Device Developer	Section
Vote multiple times using forged smartcard	•	•	•				3.2
Access administrative functions or close polling station	•	•			•	•	3.3
Modify system configuration		•			•	•	4.1
Modify ballot definition (e.g., party affiliation)		•	•	•	•	•	4.2
Cause votes to be miscounted by tampering with configuration		•	•	•	•	•	4.2
Impersonate legitimate voting machine to tallying authority		•	•	•	•	•	4.3
Create, delete, and modify votes		•	•	•	•	•	4.3, 4.5
Link voters with their votes		•	•	•	•	•	4.5
Tamper with audit logs		•			•	•	4.6
Delay the start of an election		•	•	•	•	•	4.7
Insert backdoors into code					•	•	5.3

Table 1: This table summarizes some of the more important attacks on the system.

Below we describe the process for setting up and running an election using the Diebold system. In some cases, where election procedures and policies might vary or where we have insufficient information from studying the code, we will state our assumptions. We note that, even in cases where election policies and procedures might provide protection against design shortcomings, those policies and procedures depend on poll workers who may not fully understand or be able to carry out their responsibilities. As a result, any failure in the design of the voting system may very well be abused to compromise an election.

SETTING UP. Before an election takes place, one of the first things the election officials must do is specify the political offices and issues to be decided by the voters along with the candidates and their party affiliations. Variations on the ballot can be presented to voters based on their party affiliations. We call this data a *ballot definition*. In the Diebold system, a ballot definition is encoded as the file `election.edb`.

Prior to an election, the voting terminals must be configured and installed at each voting location. A governmental entity using Diebold voting terminals has a variety of choices in how to distribute the ballot definitions. They also may be distributed using removable media, such as floppy disks or storage cards, or over a local network, the Internet, or a dial-up connection. The networked approach, if allowed under the voting precinct's processes, provides additional flexibility to the election administrator in the event of last-minute changes to the ballot.

THE ELECTION. Once the voting terminal is initialized with the ballot definitions and the election begins, voters are allowed to cast their votes. To get started, the voter must have a *voter card*. The voter card is a memory card or smartcard; i.e., it is a credit-card sized plastic card with a computer chip on it that can store data and, in the case of the smartcard, perform computation. Under the most common scenario, we assume that the voting cards are given to voters at the voting site on election day.

The voter takes the voter card and inserts it into a smartcard reader attached to the voting terminal. The terminal checks that the smartcard in its reader is a voter card and, if it is, presents a ballot to the voter on the terminal screen. The actual ballot the voter sees may depend on the voter's political party, which is encoded on the voter card. If a ballot cannot be found for the voter's party, the voter is given a nonpartisan ballot. Such party-specific ballots are used, for example, in primaries.

At this point, the voter interacts with the voting terminal, touching the appropriate boxes on the screen for his or her desired candidates. Headphones and keypads are available for visually-impaired voters to privately interact with the terminal. Before the ballots are committed to storage in the terminal, the voter is given a final chance to review his or her selections. If the voter confirms this, the vote is recorded on the voting terminal and the voter card is "canceled." This latter step is intended to prevent the voter from voting again with the same card. After the voter finishes voting, the terminal is ready for another voter to use. The voter returns his or her canceled card to the poll workers, who reprogram it for the next user.

REPORTING THE RESULTS. A poll worker ends the election process by inserting an *administrator card* or an *ender card* (a special card that can only be used to end the election) into the voting terminal. Upon detecting the presence of such a card (and, in the case of the administrator card, checking a PIN entered by the card user), the poll worker is asked to confirm that the election is finished. If the poll worker agrees, then the voting terminal enters the post-election stage. Election results are written to a removable flash memory card and can also be transmitted electronically to the back-end server.

As we have only analyzed the code for the Diebold voting terminal, we do not know exactly how the back-end server tabulates the final results it gathers from the individual terminals. Obviously, it collects all the votes from the various voting terminals. We are unable to verify that there are checks to ensure, for example, that there are no more votes collected than people who are registered at or have entered any given polling location.

DETAILED OVERVIEW OF THE CODE. The 4.3.1 snapshot of the AccuVote-TS tree [9] has 136 `.h` files totaling 16414 lines and 120 `.cpp` files totaling 33195 lines, for a total of 256 files and 49609 lines of C++

code. While a full description of every module in the Diebold AccuVote-TS 4.3.1 system is beyond the scope of this paper, we describe the bootstrapping process as well as the main state transitions that occur within a Diebold system during an election, making explicit references to the relevant portions of the code.

The voting terminal is implemented in the directory `BallotStation/`, but uses libraries in the supporting directories `Ballot/`, `DES/`, `DiagMode/`, `Shared/`, `TSElection/`, `Utilities/`, and `VoterCard/`.

The method `CBallotStationApp::DoRun()` is the main loop for the voting terminal software. The `DoRun()` method begins by invoking `CBallotStationApp::LoadRegistry()`, which loads information about the voting terminal from the registry (the registry keys are stored under `HKEY_LOCAL_MACHINE\Software\Global Election Systems\AccuVote-TS4`). If the program fails to load the registry information, it believes that it is uninitialized and therefore creates a new instance of the `CTSRegistryDlg` class that asks the administrator to set up the machine for the first time. The administrator chooses, among other things, the COM port to use with the smartcard reader, the directory locations to store files, and the polling location identifier. The `CBallotStationApp::DoRun()` method then checks for the presence of a smartcard reader and, if none is found, gives the administrator the option to interact with the `CTSRegistryDlg` again.

The `DoRun()` method then enters a `while` loop that iterates until the software is shut down. The first thing `DoRun()` does in this loop is check for the presence of some removable media on which to store election results and ballot configurations (a floppy under Windows or a removable storage card on a Windows CE device). It then tries to open the election configuration file `election.edb`. If it fails to open the configuration file, the program enters the `CTSElectionDoc::ES_NOELECTION` state and invokes `CBallotStationApp::Download()`, which creates an instance of `CTransferElecDlg` to download the configuration file. To do the download, the terminal connects to a back-end server using either the Internet or a dial-up connection. Subsequently, the program enters the `CTSElectionDoc::ES_PREELECT` state, invoking the `CBallotStationApp::PreElect()` method, which in turn creates an instance of `CPreElectDlg`. The administrator can then decide to start the election, in which case `CPreElectDlg::OnSetForElection()` sets the state of the terminal to `CTSElectionDoc::ES_ELECTION`.

Returning to the while loop in `CBallotStationApp::DoRun()`, now that the machine is in the state `CTSElectionDoc::ES_ELECTION`, the `DoRun()` method invokes `CBallotStationApp::Election()`, which creates an instance of `CVoteDlg`. When a card is inserted into the reader, the application checks to see if the card is a voter card, administrator card, or ender card. If it is an ender card, or if it is an administrator card and if the user enters the correct PIN, the `CVoteDlg` ends and the user is asked whether he or she wishes to terminate the election and, if so, the state of the terminal is set to `CTSElectionDoc::ES_POSTELECT`. If the user entered a voter card, then `DoVote()` is invoked (here `DoVote()` is an actual function; it does not belong to any class). The `DoVote()` function finds the appropriate ballot for the user's voter group or, if none exists, opens the nonpartisan ballot (recall that the system is designed to support different ballots for different voters, as might occur in a primary party election). It then creates an instance of `CBallotDlg` to display the ballot and collect the votes.

We recall that if, during the election process, someone inserted an administrator or ender card into the terminal and chooses to end the election, the system would enter the `CTSElectionDoc::ES_POSTELECT` state. At this point the voting terminal would offer the ability to upload the election results to some back-end server for final tabulation. The actual transfer of results is handled by the `CTransferResultsDlg::OnTransfer()` method.

3 Smartcards

While it is true that one can design secure systems around the use of smartcards, merely the use of smartcards in a system does *not* imply that the system is secure. The system must use the smartcards in an intelligent and security-conscious way. Unfortunately, the Diebold system's use of smartcards provides very little (if any) additional security and, in fact, opens the system to several attacks.

3.1 Exploiting the lack of cryptography: Creating homebrew smartcards

Upon reviewing the Diebold code, we observed that the smartcards do not perform any cryptographic operations. This, in and of itself, is an immediate red flag. One of the biggest advantages of smartcards over classic magnetic-stripe cards is the smartcards' ability to perform cryptographic operations internally, and with physically protected keys. Because of a lack of cryptography, *there is no secure authentication of the smartcard to the voting terminal.* This means that nothing prevents an attacker from using his or her own homebrew smartcard in a voting terminal. One might naturally wonder how easy it would be for an attacker to make such a homebrew smartcard. First, we note that user-programmable smartcards and smartcard readers are available commercially over the Internet in small quantities and at reasonable prices. Second, an attacker who knows the protocol spoken between voting terminals and legitimate smartcards could easily implement a homebrew card that speaks the same protocol. We shall shortly consider how an attacker might go about learning the protocol if he or she does not know it *a priori*.

Once the adversary knows the protocol between the terminal and the smartcards, the only impediment to the mass production of homebrew smartcards is that each voting terminal will make sure that the smartcard has encoded in it the correct `m_ElectionKey`, `m_VCenter`, and `m_DLVersion` (see `DoVote()` in `BallotStation/Vote.cpp`). The `m_ElectionKey` and `m_DLVersion` are likely the same for all locations and, furthermore, for backward-compatibility purposes it is possible to use a card with `m_ElectionKey` and `m_DLVersion` undefined. The `m_VCenter` value could be learned on a per-location-basis by interacting with legitimate smartcards, from an insider, or from inferences based on the `m_VCenter` values observed at other polling locations. In short, all the necessary information to create homebrew counterfeit smartcards is readily available.

In the next subsections we consider attacks that an adversary could mount after creating homebrew cards. We find the issues we uncovered to be particularly distressing as modern smartcard designs allow cryptographic operations to be performed directly on the smartcard, making it possible to create systems that are not as easily vulnerable to such security breaches.

REVERSE ENGINEERING THE SMARTCARD PROTOCOL. It turns out that adversaries, including regular voters, who do not know *a priori* the protocol between the smartcard and the terminal can "easily" learn the protocol, thereby allowing them to produce homebrew voter cards. An adversary, such as a poll worker, with the ability to interact with a legitimate administrator or ender card could also learn enough information to produce homebrew administrator and ender cards (Section 3.3).

Let us consider several ways that an adversary could learn the protocol between voter cards and voting terminals. After voting, instead of returning the canceled card to the poll-worker, the adversary could return a fake card that records how it is reprogrammed, and then dumps that information to a collaborating attacker waiting in line to vote. Alternatively, the attacker could attach a "wiretap" device between the voting terminal and a legitimate smartcard and observe the communicated messages. The parts for building such a device are readily available and, depending on the setup at each voting location, might be unnoticed by poll workers. An attacker might not even need to use a wiretap device: as a literal "person-in-the-middle" attack, the adversary could begin by inserting a smartcard into the terminal that records the terminal's first message. The adversary would then leave the voting location, send that message to a real voter card that he or she stole, and learn the real voter card's response. The adversary's conspirator could then show up at the

voting location and use the information gained in the first phase to learn the next round of messages in the protocol, and so on. We comment again that these techniques work because the authentication process is completely deterministic and lacks any sort of cryptography.

3.2 Casting multiple votes

In the Diebold system, a voter begins the voting process by inserting a smartcard into the voting terminal. Upon checking that the card is "active," the voting terminal collects the user's vote and then deactivates the user's card; the deactivation actually occurs by rewriting the card's type, which is stored as an 8-bit value on the card, from `VOTER_CARD` (`0x01`) to `CANCELED_CARD` (`0x08`). Since an adversary can make perfectly valid smartcards, the adversary could bring a stack of active cards to the voting booth. Doing so gives the adversary the ability to vote multiple times. More simply, instead of bringing multiple cards to the voting booth, the adversary could program a smartcard to ignore the voting terminal's deactivation command. Such an adversary could use one card to vote multiple times. Note here that the adversary could be a regular voter, and not necessarily an election insider.

Will the adversary's multiple-votes be detected by the voting system? To answer this question, we must first consider what information is encoded on the voter cards on a per-voter basis. The only per-voter information is a "voter serial number" (`m_VoterSN` in the `CVoterInfo` class). `m_VoterSN` is only recorded by the voting terminal if the voter decides *not* to place a vote (as noted in the comments in `TSElection/Results.cpp`, this field is recorded for uncounted votes for backward compatibility reasons). It is important to note that if a voter decides to cancel his or her vote, the voter will have the opportunity to vote again using that same card (and, after the vote has been cast, `m_VoterSN` will no longer be recorded).

If we assume the number of collected votes becomes greater than the number of people who showed up to vote, and if the polling locations keep accurate counts of the number of people who show up to vote, then the back-end system, if designed properly, should be able to detect the existence of counterfeit votes. However, because `m_VoterSN` is only stored for those who did not vote, there will be no way for the tabulating system to distinguish the real votes from the counterfeit votes. This would cast serious doubt on the validity of the election results. The solution proposed by one election official, to have everyone vote again, does not seem like a viable solution.

3.3 Accessing administrator and poll worker functionality

As noted in Section 2, in addition to the voter cards that normal voters use when they vote, there are also administrator cards and ender cards, which have special purposes in this system. The administrator cards give the possessor the ability to access administrative functionality (the administrative dialog `BallotStation/AdminDlg.cpp`), and both types of cards allow the possessor to end the election (hence the term "ender card").

Just as an adversary can manufacture his or her own voter cards, an adversary can manufacture his or her own administrator and ender cards (administrator cards have an easily-circumventable PIN, which we will discuss shortly). This attack is easiest if the attacker has knowledge of the Diebold code or can interact with a legitimate administrator or ender card, since otherwise the attacker would not know what distinguishes an administrator or ender card from a voter card. (The distinction is that, for a voter card `m_CardType` is set to `0x01`, for an ender card the value is `0x02`, and for an administrator card the value is `0x04`.)

As one might expect, an adversary in possession of such illicit cards has further attack options against the Diebold system. Using a homebrew administrator card, a poll worker, who might not otherwise have access to the administrator functions of the Diebold system but who does have access to the voting machines before and after the elections, could gain access to the administrator controls. If a malicious voter entered an

administrator or ender card into the voting device instead of the normal voter card, then the voter would be able to terminate the election and, if the card is an administrator card, gain access to additional administrative controls.

The use of administrator or ender cards prior to the completion of the actual election represents an interesting denial-of-service attack. Once "ended," the voting terminal will no longer accept new voters (see `CVoteDlg::OnCardIn()`) until the terminal is somehow reset. Such an attack, if mounted simultaneously by multiple people, could temporarily shut down a polling place. If a polling place is in a precinct considered to favor one candidate over another, attacking that specific polling place could benefit the less-favored candidate. Even if the poll workers were later able to resurrect the systems, the attack might succeed in deterring a large number of potential voters from voting (e.g., if the attack was performed over the lunch hour). If such an attack was mounted, one might think the attackers would be identified and caught. We note that many governmental entities, e.g., California, do not require identification to be presented by voters. By the time the poll workers realize that one of their voting terminals has been disabled, the perpetrator may have long-since left the scene. Furthermore, the poll workers may not be computer savvy and might simply think that all the machines crashed simultaneously.

CIRCUMVENTING THE ADMINISTRATOR PIN. In order to use (or create) an administrator card, the attacker must know the PIN associated (or to be associated) with the card. Because the system's use of smartcards was poorly designed, an adversary could easily learn the necessary information, thereby circumventing any security the PIN might have offered.

We first note that the PIN is sent from the smartcard to the terminal in cleartext. As a result, anyone who knows the protocol and wishes to make their own administrator card could use any PIN of their choice. Even if the attacker does not know the protocol but has access to an existing administrator card and wants to make a copy, the adversary could guess the PIN in just a few trials if the adversary realizes that the PIN is included as part of a short cleartext message sent from the card. More specifically, rather than try all 10000 possibilities for the PIN, the adversary could try all 4-byte consecutive substrings of the cleartext message.

4 Election configurations and election data

In election systems, protecting the integrity and privacy of critical data (e.g., votes, configurations, ballot definitions) is undeniably important. We investigated how the Diebold system manipulates such data, and found considerable problems. There are two main vectors for accessing and attacking the voting system's data: via physical access to the device storing the data, or via man-in-the-middle attacks as the data is transported over some network. The latter assumes that the systems are connected to a network, which is possible though may be precluded by election procedures in some jurisdictions. Attacks via physical access to memory can be quite powerful, and can be mounted easily by insiders. The network attacks, which can also be quite powerful, can also be mounted by insiders as well as sophisticated outsiders.

DATA STORAGE OVERVIEW. Each voting terminal has two distinct types of internal data storage. A main (or system) storage area contains the terminal's operating system, program executables, static data files such as fonts, and system configuration information, as well as backup copies of dynamic data files such as the voting records and audit logs. Each terminal also contains a removable flash memory storage device that is used to store the primary copies of these dynamic data files. When the terminal is running a standard copy of Windows (e.g., Windows 2000) the removable storage area is the first floppy drive; when the terminal is running Windows CE, the removable storage area is a removable storage card. Storing the dynamic data on two distinct devices is advantageous for both reliability and non-malleability: if either of the two storage mediums fails, data can still be recovered from the copy, although reconciling differences between these media may be difficult.

Unfortunately, in Windows CE, the existence of the removable storage device is not enforced properly.

Unlike other versions of Windows, removable storage cards are mounted as subdirectories under CE. When the voting software wants to know if a storage card is inserted, it simply checks to see if the `Storage Card` subdirectory exists in the filesystem's root directory. While this is the default name for a mounted storage device, it is also a perfectly legitimate directory name for a directory in the main storage area. Thus, if such a directory exists, the terminal can be fooled into using the same storage device for all of the data.[2] This would reduce the amount of redundancy in the voting system and would increase the chances that a hardware failure could cause recorded votes to be lost.

NETWORK OVERVIEW. The Diebold voting machines cannot work in isolation. They must be able to both receive a ballot definition file as input and report voting results as output. As described in Section 2, there are essentially two ways to load a voting terminal with an initial election configuration: via some removable media, such as a flash memory card, or over a network connection. In the latter case, the voting terminal could either be plugged directly into the Internet, could be connected to an isolated local network, or could use a dialup connection (the dial-up connection could be to a local ISP, or directly to the election authority's modem banks). Diebold apparently gives their customers a variety of configuration options; electronic networks are not necessary for the operation of the system. After the election is over, election results can be sent to a back-end post-processing server over the network (again, possibly through a dial-up connection). When results are reported this way, it is not clear whether these network-reported results become the official results, or just the preliminary results (the official results being computed after the memory cards are removed from all the voting terminals and collected and tabulated at a central location).

We also observe that, even in jurisdictions where voting terminals are *never* connected to a network or phone line, the physical transportation of the flash memory cards from the voting terminal to the central tabulating system is really just a "sneaker net." Such physical card transportation must be robust against real-world analogies of network man-in-the-middle attacks. Any flaws in the policies and procedures used to protect the chain of custody could lead to opportunities for these cards to be read or written by an adversary. Consequently, even if no electronic computer network is used, we still view network attacks as critical in the design of a voting system.

4.1 Tampering with the system configuration

The majority of the system configuration information for each terminal is stored in the Windows registry under `HKEY_LOCAL_MACHINE\Software\Global Election Systems\AccuVote-TS4`. This includes both identification information such as the terminal's serial number and more traditional configuration information such as the COM port to which the smartcard reader is attached. All of the configuration information is stored in the clear, without any form of integrity protection. Thus, all an adversary must do is modify the system registry to trick a given voting terminal into effectively impersonating any other voting terminal. It is unclear how the tabulating authority would deal with results from two different voting terminals with the same voting ID; at the very least, human intervention to resolve the conflict would probably be required.

The Federal Election Commission draft standard [11] requires each terminal to keep track of the total number of votes that have ever been cast on it — the "Protective Counter." This counter is used to provide yet another method for ensuring that the number of votes cast on each terminal is correct. However, as the following code from `Utilities/machine.cpp` shows, the counter is simply stored as an integer in the file `system.bin` in the terminal's system directory (error handling code has been removed for clarity):

```
long GetProtectedCounter()
```

[2] This situation can be easily corrected by checking for the `FILE_ATTRIBUTE_TEMPORARY` attribute on the directory as described in `http://msdn.microsoft.com/library/en-us/wcefiles/htm/_wcesdk_Accessing_Files_on_Other_Storage_Media.asp`.

```
{
  DWORD protectedCounter = 0;
  CString filename = ::GetSysDir();
  filename += _T("system.bin");
  CFile file;
  file.Open(filename, CFile::modeRead | CFile::modeCreate | CFile::modeNoTruncate);
  file.Read(&protectedCounter, sizeof(protectedCounter));
  file.Close();
  return protectedCounter;
}
```

We believe that the Diebold system violates the FEC requirements by storing the protected counter in a simple, mutable file. By modifying this counter, an adversary could cast doubt on an election by creating a discrepancy between the number of votes cast on a given terminal and the number of votes that are tallied in the election. While the current method of implementing the counter is totally insecure, even a cryptographic checksum would not be enough to protect the counter; an adversary with the ability to modify and view the counter would still be able to roll it back to a previous state. In fact, the only solution that would work would be to implement the protective counter in a tamper-resistant hardware token, but doing so would require physical modifications to existing hardware.

4.2 Tampering with ballot definitions

The "ballot definition" for each election (`election.edb`) contains everything from the background color of the screen and information about the candidates and issues on the ballot to the PPP username and password to use when reporting the results, if reporting the results over a dial-up connection. This data is neither encrypted nor checksummed (cryptographically or otherwise).

If uninterrupted physical access is *ever* available to the voting terminal after the ballot definition has been loaded, perhaps the night before an election, using a janitor's master keys to the building, then it would be possible for an adversary to tamper with the voting terminals' ballot definition file or to even tamper with the voting software itself. Protections such as physical locks or tamper-evident seals may somewhat allay these concerns, but we would prefer designs that can be robust even against physical tampering.

On a potentially much larger scale, if the voting terminals download the ballot definition over a network connection, then an adversary could tamper with the ballot definition file en-route from the back-end server to the voting terminal; of course, additional poll-worker procedures could be put in place to check the contents of the file after downloading, but we prefer a technological solution. With respect to modifying the file as it is sent over a network, we point out that the adversary need not be an election insider; the adversary could, for example, be someone working at the local ISP. If the adversary knows the structure of the ballot definition, then the adversary can intercept and modify the ballot definition while it is being transmitted. Even if the adversary does not know the precise structure of the ballot definition, many of the fields inside are easy to identify and change, including the candidates' names, which appear as plain ASCII text.

Because no cryptographic techniques are in place to guard the integrity of the ballot definition file, an attacker could add, remove, or change issues on the ballot, and thereby confuse the result of the election. In the system, different voters can be presented with different ballots depending on their party affiliations (see `CBallotRelSet::Open()`, which adds different issues to the ballot depending on the voter's `m_VGroup1` and `m_VGroup2` `CVoterInfo` fields). If an attacker changes the party affiliations of the candidates, then he may succeed in forcing the voters to view and vote on erroneous ballots.[3] More subtle

[3] As an example of what might happen if the party affiliations were listed incorrectly, we note that, according to a news story at `http://www.gcn.com/vol19_no33/news/3307-1.html`, in the 2000 New Mexico presidential election, over 65,000 votes were incorrectly counted because a worker accidentally had the party affiliations wrong. (We are not claiming this worker had malicious intent, nor are we implying that this error had an effect on the results of the election.)

attacks are also possible. By simply changing the order of the candidates as they appear in the ballot definition, the results file will change accordingly. However, the candidate information itself is not stored in the results file, which merely tracks that candidate 1 got so many votes and candidate 2 got so many other votes. If an attacker reordered the candidates on the ballot definition, voters would unwittingly cast their ballots for the wrong candidate. Ballot reordering attacks would be particularly effective in polling locations known to have more voters of one party than another. (In Section 4.3 and Section 4.5 we consider other ways of tampering with the election results.)

4.3 Impersonating legitimate voting terminals

Consider voting terminals that are configured to upload voting totals to some back-end tabulating authority after an election. An adversary able to pose as a legitimate voting terminal to the tabulating authority could obviously cause (at least temporary) damage by reporting false vote counts to the tabulating system. If the voting terminals use a normal Internet connection, then an adversary with the ability to sniff the connection of a legitimate terminal could learn enough information (e.g., the IP address of the back-end server) to be able to impersonate a legitimate terminal. If the terminals use a dialup connection, then the adversary would either need to be able to sniff a legitimate dialup connection to learn the appropriate information (e.g., the dial-up PPP number, login, and password), or must garner that information in another way. The PPP phone number, username, password, and IP address of the back-end server are stored in the registry `HKEY_LOCAL_MACHINE\Software\Global Election Systems\AccuVote-TS4\TransferParams`, thus making it easily accessible to an insider working at the polling station. By studying the configuration of the ballot definition files, we learned that the definition files also store the terminal's voting center ID, PPP dial-in number, username, password and the IP address of the back-end server (these are parsed into a `CElectionHeaderItem` in `TSElection\TSElectionObj.cpp`). The ballot definition files thus provide another vector for an adversary to learn almost all of the information necessary to impersonate a real voting terminal over a dialup connection (the adversary would also have to create a voting terminal ID, although the ID may or may not be checked for legitimacy by the back-end server).

4.4 Key management and other cryptographic issues with the vote and audit records

Unlike the other data stored on the voting terminal, both the vote records and the audit logs are encrypted and checksummed before being written to the storage device. Unfortunately, neither the encrypting nor the checksumming is done with established, secure techniques. This section summarizes the issues with Diebold's use of cryptography in protecting the vote records and audit logs, and then return to consequences of Diebold's poor choices in subsequent subsections. (Recall that we have already discussed the lack of cryptography in other potions of the system.)

KEY MANAGEMENT. All of the data on a storage device is encrypted using a single, hardcoded DES [22] key:

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

Note that this value is not a hex representation of a key, nor does it appear to be randomly generated. Instead, the bytes in the string "`F2654hD4`" are fed directly into the DES key scheduler. It is well-known that hard-coding keys into a program's source code is a bad idea: if the same compiled program image is used on every voting terminal, an attacker with access to the source code, or even to a single program image, could learn the key and thus read and modify voting and auditing records. The case with the Diebold system is even worse: from the CVS logs, we see this particular key has been used without change since December 1998, when the CVS tree for AccuVote-TS version 3 began, and we assume that the key was in use much before

that. Although Jones reports that the vendor may have been aware of the key management problems in their code since at least 1997 [16, 17], our findings show that the design flaw was never addressed. The SAIC analysis of Diebold's system [27] agrees that Diebold needs to redesign their cryptography architecture. The most appropriate solution will likely involve the use of hardware cryptographic coprocessors.

(In a similar fashion, Diebold's voter, administrator, and ender cards use a hardcoded 8-byte password `ED 0A ED 0A ED 0A ED 0A` (hexadecimal) to authenticate the voting terminals to the smartcards, transmitted in cleartext. The smartcards are discussed in Section 3.)

"ENCRYPTION." Even if proper key management were to be implemented, however, many problems would still remain. First, DES keys can be recovered by brute force in a very short time period [12]. DES should be replaced with either triple-DES [26] or, preferably, AES [8]. Second, DES is being used in CBC mode which requires a random initialization vector to ensure its security. The implementation here always uses zero for its IV. This is illustrated by the call to `DesCBCEncrypt` in `TSElection/RecordFile.cpp`; since the second to last argument is NULL, `DesCBCEncrypt` will use the all-zero IV.

```
DesCBCEncrypt((des_c_block*)tmp, (des_c_block*)record.m_Data, totalSize,
              DESKEY, NULL, DES_ENCRYPT);
```

To correctly implement CBC mode, a source of "strong" random numbers must be used to generate a fresh IV for each encryption [2]. Suitably strong random numbers can be derived from many different sources, ranging from custom hardware to accumulated observations of user behavior.

"MESSAGE AUTHENTICATION." Before being encrypted, a 16-bit cyclic redundancy check (CRC) of the plaintext data is computed. This CRC is then stored along with the ciphertext in the file and verified whenever the data is decrypted and read. This process in handled by the `ReadRecord` and `WriteRecord` functions in `TSElection/RecordFile.cpp`. Since the CRC is an unkeyed, public function, it does not provide any meaningful integrity protection for the data. In fact, by storing it in an unencrypted form, the purpose of encrypting the data in the first place (leaking no information about the contents of the plaintext) is undermined. Standard industry practice would be to first encrypt the data to be stored and then to compute a keyed cryptographic checksum (such as HMAC-SHA1 [1]) of the ciphertext [3, 19]. This cryptographic checksum could then be used to detect any tampering with the plaintext. Note also that each entry has a timestamp, which can be used to detect reordering, although sequence numbers should also be added to detect record deletion.

4.5 Tampering with election results and linking voters with their votes

A likely attack target are the voting records themselves. When stored on the device, the voting records are "encrypted" as described in Section 4.4. If the votes are transmitted to a back-end authority over a network connection, as appears to be the case in at least some areas, no cryptography is used: the votes are sent in cleartext. In particular, `CTransferResultsDlg::OnTransfer()` writes ballot results to an instance of `CDL2Archive`, which then writes the votes in cleartext to a socket without any cryptographic checksum. If the network connection is via a cable modem or a dedicated connection, then the adversary could be an employee at the local ISP. If the voting terminals use a dialup connection directly to the tabulating authority's network, then the risk of such an attack is less, although still not inconsequential. A sophisticated adversary, e.g., an employee of the local phone company, could tap the phone line and intercept the communication.

TAMPERING WITH ELECTION RESULTS. In Section 4.2 we showed that an adversary could alter election results by modifying ballot definition files, and in Section 4.3 we showed that an adversary could inject fake votes to a back-end tabulating authority by impersonating a legitimate voting terminal. Here we suggest another way to modify the election result: modify the voting records file stored on the device. Because of the poor cryptography described in Section 4.4, an attacker with access to this file would be able to

generate or change as many votes as he or she pleased. Furthermore, the adversary's modified votes would be indistinguishable from the true votes cast on the terminal. The attack described here is more advantageous to an adversary than the attacks in Section 4.2 and Section 4.3 because it leaves no evidence that an attack was ever mounted (whereas the attacks in Section 4.2 and Section 4.3 could be discovered but not necessarily corrected as part of a post-election auditing phase).

If the votes are sent to the back-end authority over a network, then there is another vector for an adversary to modify the election results. Specifically, an adversary with the ability to tamper with the channel could introduce new votes or modify existing votes. Such an attacker could, for example, decrease one candidate's vote count by some number while increasing another's candidate's count by the same number. Of course, to introduce controlled changes such as these to the votes, the attacker would benefit from some knowledge of the structure of the protocol used between the terminals and the back-end server. This form of tampering might later be detected by comparing the memory storage cards to data transmitted across the networks, although the memory storage cards themselves might also be subject to tampering. (We briefly comment that these network attacks could be largely circumvented with the use of standard cryptographic tools, such as SSL/TLS.)

LINKING VOTERS WITH THEIR VOTES. From analyzing the code, we learned that each vote is written *sequentially* to the file recording the votes. This fact provides an easy mechanism for an attacker, such as a poll worker with access to the voting records, to link voters with their votes. A poll worker could surreptitiously track the order in which voters use the voting terminals. Later, in collaboration with other attackers who might intercept the "encrypted" voting records, the exact voting record of each voter could be reconstructed.

If the results are transmitted over a network, as is the case in at least some jurisdictions, then physical access to the voting results is not even necessary. Recall that, when transmitted over the network, the votes are sent in unencrypted, cleartext form.

"RANDOMIZED" SERIAL NUMBERS. While the voter's identity is not stored with the votes, each vote is given a serial number in order to "randomize" the votes after they are uploaded to the back-end tabulating authority. As we noted above, randomizing the order of votes *after* they are uploaded to the the tabulating authority does not prevent the possibility of linking voters to their votes. Nevertheless, it appears that the designers wanted to use a cryptographically secure pseudorandom number generator to generate serial numbers for some post-processing purposes. Unfortunately, the pseudorandom number generator they chose to use (a linear congruential generator) is not cryptographically secure. Moreover, the generator is seeded with static information about the voting terminal and the election.

```
// LCG - Linear Conguential Generator - used to generate ballot serial numbers
// A psuedo-random-sequence generator
// (per Applied Cryptography, by Bruce Schneier, Wiley, 1996)
#define LCG_MULTIPLIER 1366
#define LCG_INCREMENTOR 150889
#define LCG_PERIOD 714025

static inline int lcgGenerator(int lastSN)
{
  return ::mod(((lastSN * LCG_MULTIPLIER) + LCG_INCREMENTOR), LCG_PERIOD);
}
```

It is interesting to note that the code's authors apparently decided to use an linear congruential generator because it appeared in *Applied Cryptography* [26] even though in the same work it is advised that such generators should not be used for cryptographic purposes.

4.6 Audit logs

Each entry in a plaintext audit log is simply a timestamped, informational text string. There appears to be no clear pattern for what is logged and what is not. The whole audit log is encrypted using the insecure method described in Section 4.4. An adversary with access to the audit log file could easily change its contents.

At the time that the logging occurs, the log can also be printed to an attached printer. If the printer is unplugged, off, or malfunctioning, no record will be stored elsewhere to indicate that the failure occurred. The following code from `TSElection/Audit.cpp` demonstrates that the designers failed to consider these issues:

```
if (m_Print && print) {
  CPrinter printer;
  // If failed to open printer then just return.
  CString name = ::GetPrinterPort();
  if (name.Find(_T("\\")) != -1)
    name = GetParentDir(name) + _T("audit.log");
  if (!printer.Open(name, ::GetPrintReverse(), FALSE))
    ::TSMessageBox(_T("Failed to open printer for logging"));
  else {
    [ do the printing ]
  }
}
```

If the cable attaching the printer to the terminal is exposed, an attacker could create discrepancies between the printed log and the log stored on the terminal by unplugging the printer (or, by simply cutting the cable).

4.7 Attacking the start of an election

Although good election processes would dictate installing the ballot definition files well before the start of the election, we can imagine scenarios in which the election officials must reinstall ballot files shortly before the start of an election, and do not have time to distribute the definition files manually.[4]

One option for the election officials would be to download the files over the Internet. In addition to the problems we have outlined, we caution against relying on such an approach, as an adversary could mount a traditional Internet denial-of-service attack against the election management's server and thereby prevent the voting terminals from acquiring their ballot definition files in time for the election. Even a general idea of the range of Internet addresses used by the election administration would be sufficient for an attacker to target a large-scale distributed denial of service (DDoS) attack.

Of course, we acknowledge that there are other ways to postpone the start of an election at a voting location that do not depend on Internet DDoS attacks (e.g., flat tires for all poll workers for a given precinct, or other acts of real-world vandalism). Unlike such traditional attacks, however, (1) the network-based attack is relatively easy for anyone with knowledge of the election system's network topology to accomplish; (2) this attack can be performed on a very large scale, as the central distribution point(s) for ballot definitions becomes an effective single point of failure; and (3) the attacker can be physically located anywhere in the Internet-connected world, complicating efforts to apprehend the attacker. Such attacks could prevent or delay the start of an election at all voting locations in a state. We note that this attack is not restricted to the system we analyzed; it is applicable to any system that downloads its ballot definition files using the Internet or otherwise relies upon the Internet.

[4] In recent elections, we have seen cases where politicians passed away or withdrew from the race very close to the election day.

5 Software engineering

When creating a secure system, getting the design right is only part of the battle. The design must then be securely implemented. We now examine the coding practices and implementation style used to create the voting system. This type of analysis can offer insights into future versions of the code. For example, if a current implementation has followed good implementation practices but is simply incomplete, one would be more inclined to believe that future, more complete versions of the code would be of a similar high quality. Of course, the opposite is also true, perhaps even more so: it is very difficult to produce a secure system by building on an insecure foundation.

Of course, reading the source code to a product gives only an incomplete view into the actions and intentions of the developers who created that code. Regardless, we can see the overall software design, we can read the comments in the code, and, thanks to the CVS repository, we can even look at earlier versions of the code and read the developers' commentary as they committed their changes to the archive.

5.1 Code legacy

Inside `cvs.tar` we found multiple CVS archives. Two of the archives, `AccuTouch` and `AVTSCE`, implement full voting terminals. The `AccuTouch` code, corresponding to AccuVote-TS version 3, dates from December 1998 to August 2001 and is copyrighted by "Global Election Systems, Inc.," while the `AVTSCE` code, corresponding to the AccuVote-TS version 4 system, dates from October 2000 to April 2002 and is copyrighted by "Diebold Election Systems, Inc." (Diebold acquired Global Election Systems in September 2001.[5]) Although the `AccuTouch` tree is not an immediate ancestor of the `AVTSCE` tree (from the CVS logs, the `AVTSCE` tree is actually an import of another `AccuTouch-CE` tree that we do not have), the `AccuTouch` and `AVTSCE` trees are related, sharing a similar overall design and a few identical files. From the comments, some of the code, such as the functions to compute CRCs and DES, date back to 1996 and a company later acquired by Global Election Systems called "I-Mark Systems." We have already remarked (Section 4.4) that the same DES key has been hardcoded into the system since at least the beginning of the `AccuTouch` tree.

5.2 Coding style

While the system is implemented in an unsafe language[6] (C++), the code reflects an awareness of avoiding such common hazards as buffer overflows. Most string operations already use their safe equivalents, and there are comments, e.g., `should really use _snprintf`, reminding the developers to change others. While we are not prepared to claim that there are no exploitable buffer overflows in the current code, there are at the very least no glaringly obvious ones. Of course, a better solution would have been to write the entire system in a safe language, such as Java or Cyclone [15]. In such a language we would be able to prove that large classes of attacks, including buffer overflows and type-confusion attacks, are impossible assuming a correct implementation of the compiler and runtime system.

Overall, the code is rather unevenly commented. While most files have a description of their overall function, the meanings of individual functions, their arguments, and the algorithms within are more often than not undocumented. An example of a complex and completely undocumented function is the `CBallotRelSet::Open` function from `TSElection/TSElectionSet.cpp` as shown in Figure 2. This block of code contains two nested loops, four complex conditionals, and five debugging assertions, but no comments that explain its purpose. Ascertaining the meaning of even a small part of this code is a huge undertaking. For example, what does it mean for `vgroup->KeyId() == -1`? That the ID is simply

[5] `http://dallas.bizjournals.com/dallas/stories/2001/09/10/daily2.html`

[6] Here we mean language safety in the technical sense: no primitive operation in any program ever misinterprets data.

```
void CBallotRelSet::Open(const CDistrict* district, const CBaseunit* baseunit,
                                      const CVGroup* vgroup1, const CVGroup* vgroup2)
{
   ASSERT(m_pDB != NULL);
   ASSERT(m_pDB->IsOpen());
   ASSERT(GetSize() == 0);
   ASSERT(district != NULL);
   ASSERT(baseunit != NULL);

   if (district->KeyId() == -1) {
      Open(baseunit, vgroup1);
   } else {
      const CDistrictItem* pDistrictItem = m_pDB->Find(*district);
      if (pDistrictItem != NULL) {
         const CBaseunitKeyTable& baseunitTable = pDistrictItem->m_BaseunitKeyTable;
         int count = baseunitTable.GetSize();
         for (int i = 0; i < count; i++) {
            const CBaseunit& curBaseunit = baseunitTable.GetAt(i);
            if (baseunit->KeyId() == -1 || *baseunit == curBaseunit) {
               const CBallotRelationshipItem* pBalRelItem = NULL;
               while ((pBalRelItem = m_pDB->FindNextBalRel(curBaseunit, pBalRelItem))){
                  if (!vgroup1 || vgroup1->KeyId() == -1 ||
                    (*vgroup1 == pBalRelItem->m_VGroup1 && !vgroup2) ||
                    (vgroup2 && *vgroup2 == pBalRelItem->m_VGroup2 &&
                       *vgroup1 == pBalRelItem->m_VGroup1))
                     Add(pBalRelItem);
               }
            }
         }
         m_CurIndex = 0;
         m_Open = TRUE;
      }
   }
}
```

Figure 2: The function `CBallotRelSet::Open` function from `TSElection/TSElectionSet.cpp`. This complex function is completely undocumented.

---

undefined? Or perhaps that the group should be ignored? Such poorly documented code impairs the ability of both internal developers and external security evaluator to assess whether the code is functioning properly or might lead to a security issue.

5.3 Coding process

An important point to consider is how code is added to the system. From the project's CVS logs, we can see that most recent code updates are in response to specific bugs that needed to be fixed. There are, however, no references to tracking numbers from a bug database or any other indication that such fixes have been vetted through any change-control process. Indeed, each of the programmers[7] seem to have completely autonomous authority to commit to any module in the project. The only evidence that we have found that the code undergoes any sort of review comes from a single log comment: "Modify code to avoid multiple exit points to meet Wyle requirements." This refers to Wyle Labs, one of the independent testing authorities charged with certifying that voting machines have met FEC guidelines.

Virtually any serious software engineering endeavor will have extensive design documents that specify how the system functions, with detailed descriptions of all aspects of the system, ranging from the user interfaces through the algorithms and software architecture used at a low level. We found no such documents in the CVS archive, and we also found no references to any such documents in the source code, despite references to algorithms textbooks and other external sources.

There are also pieces of the voting system that come from third parties. Most obviously, a flaw in the operating system, Windows CE, could expose the system to attack since the OS controls memory manage-

[7]Through web searches, we have matched each programmer's CVS user names with their likely identities and so can conclude that they are not group accounts.

ment and all of the device's I/O needs. In addition, an audio library called *fmod* is used.[8] While the source to *fmod* is available with commercial licenses, unless this code is fully audited it might contain a backdoor or an exploitable buffer overflow. Since both the operating system and *fmod* can access the memory of the voting program, both must be considered part of the trusted computing base (TCB) as a security vulnerability in either could compromise the security of the voting program itself. The voting terminal's hardware boot instructions should likewise be considered part of the TCB.

Due to the lack of comments, the legacy nature of the code, and the use of third-party code and operating systems, we believe that any sort of comprehensive, top-to-bottom code review would be nearly impossible. Not only does this increase the chances that bugs exist in the code, but it also implies that any of the coders could insert a malicious backdoor into the system without necessarily being caught. The current design deficiencies provide enough other attack vectors, however, that such an explicit backdoor is not required to successfully attack the system. Regardless, even if the design problems are eventually rectified, the problems with the coding process may well remain intact.

Since the initial version of this paper was made available on the Internet, Diebold has apparently "developed, documented, and implemented a change control process" [27]. The details of this revised process have not been made available to the public, so we are unable to comment on their effectiveness.

5.4 Code completeness and correctness

While the code we studied implements a full system, the implementors have included extensive comments on the changes that would be necessary before the system should be considered complete. It is unclear whether the programmers actually intended to go back and remedy all of these issues as many of the comments existed, unchanged, for months, while other modifications took place around them. Of course, while the AVTSCE code we examined appears to have been the current codebase in April 2002, we know nothing about subsequent changes to the code. (Modification dates and locations are easily visible from the CVS logs.) These comments come in a number of varieties. For illustrative purposes, we have chosen to show a few such comments from the subsystem that plays audio prompts to visually-impaired voters.

- Notes on code reorganization:
  ```
  /* Okay, I don't like this one bit.  Its really tough to tell where m_AudioPlayer
  should live.  [...]  A reorganization might be in order here.  */
  ```
- Notes on parts of code that need cleaning up:
  ```
  /* This is a bit of a hack for now.  [...]  Calling from the timer message
  appears to work.  Solution is to always do a 1ms wait between audio clips.  */
  ```
- Notes on bugs that need fixing:
  ```
  /* need to work on exception *caused by audio*.  I think they will currently
  result in double-fault.  */
  ```

There are, however, no comments that would suggest that the design will radically change from a security perspective. None of the security issues that have been discussed in this paper are pointed out or marked for correction. In fact, the only evidence at all that a redesign might at one point have been considered comes from outside the code: the Crypto++ library[9] is included in another CVS archive in `cvs.tar`. However, the library was added in September 2000, before the start of the AVTSCE AccuVote-TS version 4 tree, and appears to have never been used. (The subsequent SAIC [27] and RABA [24] analyses report that many of the problems we identify are still applicable to recent versions of the AccuVote-TS system, implying

[8] `http://www.fmod.org/`

[9] `http://www.eskimo.com/~weidai/cryptlib.html`

that, at least up to the version that SAIC and RABA analyzed, there has not been any radical change to the AccuVote-TS system.)

6 Conclusions

Using publicly available source code, we performed an analysis of the April 2002 snapshot of Diebold's AccuVote-TS 4.3.1 electronic voting system. We found significant security flaws: voters can trivially cast multiple ballots with no built-in traceability, administrative functions can be performed by regular voters, and the threats posed by insiders such as poll workers, software developers, and janitors is even greater. Based on our analysis of the development environment, including change logs and comments, we believe that an appropriate level of programming discipline for a project such as this was not maintained. In fact, there appears to have been little quality control in the process.

For quite some time, voting equipment vendors have maintained that their systems are secure, and that the closed-source nature makes them even more secure. Our glimpse into the code of such a system reveals that there is little difference in the way code is developed for voting machines relative to other commercial endeavors. In fact, we believe that an open process would result in more careful development, as more scientists, software engineers, political activists, and others who value their democracy would be paying attention to the quality of the software that is used for their elections. (Of course, open source would not solve all of the problems with electronic elections. It is still important to verify somehow that the binary program images running in the machine correspond to the source code and that the compilers used on the source code are non-malicious. However, open source is a good start.) Such open design processes have proven successful in projects ranging from very focused efforts, such as specifying the Advanced Encryption Standard (AES) [23], through very large and complex systems such as maintaining the Linux operating system. Australia is currently using an open source voting system[10].

Alternatively, security models such as the voter-verified audit trail allow for electronic voting systems that produce a paper trail that can be seen and verified by a voter. In such a system, the correctness burden on the voting terminal's code is significantly less as voters can see and verify a physical object that describes their vote. Even if, for whatever reason, the machines cannot name the winner of an election, then the paper ballots can be recounted, either mechanically or manually, to gain progressively more accurate election results. Voter-verifiable audit trails are required in some U.S. states, and major DRE vendors have made public statements that they would support such features if their customers required it. The EVM project[11] is an ambitious attempt to create an open-source voting system with a voter-verifiable audit trail — a laudable goal.

The model where individual vendors write proprietary code to run our elections appears to be unreliable, and if we do not change the process of designing our voting systems, we will have no confidence that our election results will reflect the will of the electorate. We owe it to ourselves and to our future to have robust, well-designed election systems to preserve the bedrock of our democracy.

Acknowledgments

We thank Cindy Cohn, David Dill, Badri Natarajan, Jason Schultz, Tracy Volz, David Wagner, and Richard Wiebe for their suggestions and advice. We also thank the state of Maryland for hiring SAIC and RABA and the state of Ohio for hiring Compuware to independently validate our findings.

[10] `http://www.elections.act.gov.au/EVACS.html`

[11] `http://evm2003.sourceforge.net`

References

[1] M. Bellare, R. Canetti, and H. Krawczyk. Keying hash functions for message authentication. In N. Koblitz, editor, *Advances in Cryptology – CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, Berlin Germany, Aug. 1996.

[2] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 394–403. IEEE Computer Society Press, 1997.

[3] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In T. Okamoto, editor, *Advances in Cryptology – ASIACRYPT 2000*, volume 1976 of *Lecture Notes in Computer Science*, pages 531–545. Springer-Verlag, Berlin Germany, Dec. 2000.

[4] California Internet Voting Task Force. *A Report on the Feasibility of Internet Voting*, Jan. 2000. `http://www.ss.ca.gov/executive/ivote/`.

[5] *Voting: What Is; What Could Be*, July 2001. `http://www.vote.caltech.edu/Reports/`.

[6] D. Chaum. Secret-ballot receipts: True voter-verifiable elections. *IEEE Security and Privacy*, 2(1):38–47, 2004.

[7] Compuware Corporation. *Direct Recording Electronic (DRE) Technical Security Assessment Report*, Nov. 2003. `http://www.sos.state.oh.us/sos/hava/files/compuware.pdf`.

[8] J. Daemen and V. Rijmen. *The Design of Rijndael: AES–The Advanced Encryption Standard*. Springer-Verlag, Berlin Germany, 2002.

[9] Diebold Election Systems. `AVTSCE` source tree, 2003. `http://users.actrix.co.nz/dolly/Vol2/cvs.tar`.[12]

[10] D. L. Dill, R. Mercuri, P. G. Neumann, and D. S. Wallach. *Frequently Asked Questions about DRE Voting Systems*, Feb. 2003. `http://www.verifiedvoting.org/drefaq.asp`.

[11] Federal Election Commission. *Voting System Standards*, 2001. `http://fecweb1.fec.gov/pages/vss/vss.html`.

[12] J. Gilmore, editor. *Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design*. O'Reilly, July 1998.

[13] D. Gritzalis, editor. *Secure Electronic Voting*. Springer-Verlag, Berlin Germany, 2003.

[14] B. Harris. *Black Box Voting: Vote Tampering in the 21st Century*. Elon House/Plan Nine, July 2003.

[15] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, June 2002.

[16] D. W. Jones. *Problems with Voting Systems and the Applicable Standards*, May 2001. Testimony before the U.S. House of Representatives' Committee on Science, `http://www.cs.uiowa.edu/~jones/voting/congress.html`.

[12] The `cvs.tar` file has been removed from this website.

[17] D. W. Jones. *The Case of the Diebold FTP Site*, July 2003. `http://www.cs.uiowa.edu/~jones/voting/dieboldftp.html`.

[18] A. Kerckhoffs. *La Cryptographie Militaire*. Libraire Militaire de L. Baudoin & Cie, Paris, 1883.

[19] H. Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In J. Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 310–331. Springer-Verlag, Berlin Germany, 2001.

[20] R. Mercuri. *Electronic Vote Tabulation Checks and Balances*. PhD thesis, University of Pennsylvania, Philadelphia, PA, Oct. 2000.

[21] National Science Foundation. *Report on the National Workshop on Internet Voting: Issues and Research Agenda*, Mar. 2001. `http://news.findlaw.com/cnn/docs/voting/nsfe-voterprt.pdf`.

[22] NBS. Data encryption standard, January 1977. Federal Information Processing Standards Publication 46.

[23] J. Nechvatal, E. Barker, L. Bassham, W. Burr, M. Dworkin, J. Foti, and E. Roback. *Report on the Development of the Advanced Encryption Standard (AES)*, Oct. 2000.

[24] RABA Innovative Solution Cell. *Trusted Agent Report: Diebold AccuVote-TS Voting System*, Jan. 2004. `http://www.raba.com/press/TA_Report_AccuVote.pdf`.

[25] A. D. Rubin. Security considerations for remote electronic voting. *Communications of the ACM*, 45(12):39–44, Dec. 2002. `http://avirubin.com/e-voting.security.html`.

[26] B. Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, New York, second edition, 1996.

[27] Science Applications International Corporation. *Risk Assessment Report: Diebold AccuVote-TS Voting System and Processes*, Sept. 2003. `http://www.dbm.maryland.gov/SBE`.