Electronic voting with Scantegrity: analysis and exposing a vulnerability

John S. Dean

Department of Computer Science,
Information Systems, and Mathematics,
Park University,
8700 N.W. River Park Drive,
Parkville, MO 64152, USA
E-mail: john.dean@park.edu

Abstract: This paper describes Scantegrity, an electronic voting system developed by Chaum et al. (2008b), and it analyses Scantegrity’s ability to satisfy the goals of privacy and verifiability. The paper describes a programmatic attack on Scantegrity and presents findings in the form of program output for a corrupted hypothetical election. The attack takes advantage of the inherent vulnerability of electronic voting systems due to the tension between the goals of privacy and verifiability. It exposes a security weakness in electronic voting systems, and it establishes the need to control code compilation and provide physical security for that compiled code throughout the election process.

Keywords: Scantegrity; e-government; e-voting; mix network; RPC; randomised partial checking; attack.


Biographical notes: John S. Dean is an Assistant Professor and the Department Chair of the Computer Science, Information Systems, and Mathematics Department at Park University, located in Parkville, Missouri. He received his Master’s in Computer Science from the University of Kansas and is currently working on his PhD in Computer Science at Nova Southeastern University. He is the co-author of Introduction to Programming with Java – A Problem Solving Approach (McGraw-Hill, 2008).

1 Introduction

Since voting is one of the fundamental principles upon which many governments and other organisations are founded, there is a great deal of interest in ensuring that electronic voting systems are at least as fair as manual voting systems. This paper addresses that interest. In Mercuri and Neumann (2003b) stated that there is a commonly held view that electronic elections (as opposed to manual elections) lead to improvements in terms of the accuracy and speed of vote counts. Presumably, such improvements would be due to the elimination of slow and accident-prone human counting. Electronic elections have
been shown to improve the vote counting speed, but it’s unclear as to whether they have also led to improvements in vote counting accuracy.

Despite ongoing concerns (Kohno et al., 2004; Morozov, 2009), there is great interest and progress in the area of electronic voting. In the USA, some state counties use manual voting machines, but the vast majority of counties use electronic voting machines that read votes with optical scanners or touch screens and count the votes electronically (Fischetti, 2008). According to Smith (2007), there is pressure on governments to move to electronic voting in order to keep up with the trends that show an increase in electronic commerce. However, there is still much debate as to the efficacy of electronic voting – whether its potential improvements in ease of use and speed are more than offset by potential problems.

In the upcoming Section 2, I will present various strategies that have been used to address the concerns over electronic voting systems. In particular, I will discuss how electronic voting systems have attempted to further the goals of privacy and verifiability. Different implementation strategies will be mentioned, but the primary focus will be in the context of explaining Scantegrity. Scantegrity was chosen as this paper’s primary focus because in November, 2009, it became the first end-to-end verifiable electronic voting system to be used in a binding secret-ballot election for public office (Carback et al., 2010). An end-to-end verifiable voting system is one in which the voter can verify that his or her vote was recorded correctly and that all the votes were tallied correctly.

For its research question, this paper attempts to determine whether an attack can be devised to compromise the results of a Scantegrity-run election. In a study that interviewed voters using an electronic voting system (Chiang, 2009), it was found that if a voting system’s security is considered to be low, then voters’ trust and their willingness to accept the voting outcome decreases. This implies that if an attack can be devised for Scantegrity, the knowledge of such an attack could play a significant role in the success of Scantegrity, even if the attack is never used.

In Section 3, I will describe a proposed attack on the Scantegrity system which takes advantage of the inherent tension between the need to satisfy both the privacy goal and the verifiability goal. In Section 4, I will discuss the effectiveness of the attack and introduce a strategy for thwarting the attack. In addition, Also in Section 4, I will introduce a general technique which can be used to verify, with a minimum probability, that a particular Scantegrity election is correct.

1.1 Secure electronic voting goals

There is no canonical list of goals for secure electronic elections, but there is general agreement that the following issues are central to the design of secure electronic elections (Delaune et al., 2009; Lambrinoudakis et al., 2003; Magkos et al., 2007):

- **Eligibility**: Only registered voters can vote, and just one vote per voter.
- **Accuracy**: Votes are to be recorded and tallied correctly.
- **Fairness**: Vote tallies are to remain secret until all voting has finished.
- **Vote privacy**: There is no publicly available information that ties voters to their votes.
Electronic voting with Scantegrity: analysis and exposing a vulnerability

- **Coercion resistance**: It should be impossible for voters to prove (e.g., with a receipt) who they voted for. With no proof, voters cannot be coerced into revealing their votes to an adversary.

- **Individual verifiability**: Voters can confirm that their votes were recorded correctly.

- **Universal verifiability**: Anyone can confirm that all the votes were tallied correctly.

Privacy and verifiability are often considered to be contradictory goals (Lambrinoudakis, et al., 2003). The privacy goal requires that a voter’s vote cannot be traced back to the voter. On the other hand, the verifiability goal requires that the voter be able to verify that his or her vote was recorded correctly. Such verification would be easy if voter names and their votes were publicly displayed, but that would clearly violate the privacy goal.

In the interest of narrowing the paper’s focus, this paper will not address issues involving eligibility and fairness. It will concentrate on just the other goals listed above – vote privacy, coercion resistance, individual verifiability, universal verifiability, and accuracy.

In the interest of narrowing this paper’s focus, the paper will not address issues involving online voting such as votes being intercepted on the Internet and over-the-shoulder vote monitoring by coercers. This paper will limit security issues to those that arise when eligible voters vote at polling stations.

2 Literature review

2.1 Scantegrity overview

Chaum et al. (2008a, 2008b) described an electronic voting system, Scantegrity, that claims to satisfy all of the secure election goals mentioned above except for eligibility. It handles security issues that arise when eligible voters vote at a polling station. The Scantegrity system (often referred to as Scantegrity II, where ‘II’ stands for invisible ink) attempts to implement an end-to-end voting scheme where voter privacy is preserved and there exists public verifiability all the way from the voters to the resulting tallies. That sort of scheme is referred to as chain-of-custody confirmation.

Scantegrity attempts to satisfy the goals of individual verifiability, where individual voters can confirm that their votes were recorded correctly, and universal verifiability, where anyone can verify that the cast votes were correctly tallied for each candidate. Such verifications are an improvement over manual voting systems, where voting authorities, not public individuals, are in charge of verifying that votes were recorded correctly and that cast votes were tallied correctly (Chaum et al., 2008a).

In Mercuri and Neumann (2003b) criticised voting systems that were purely electronic. They strongly advocated the need for paper ballots and paper receipts so that voters can more easily verify that their votes are recorded and counted correctly. Their primary concern was that without paper ballots and paper receipts, electronic voting systems did a poor job of providing vote verifiability. In Mercuri (2002) acknowledged that Chaum’s contemporary electronic voting system (Chaum, 2004) was an improvement in secure electronic elections because it used paper ballots and ballot receipts. Chaum’s 2008 Scantegrity system is the successor to that electronic voting system, and it also uses paper ballots and paper receipts.
Even though it’s a work in progress, Scantegrity has already been used in a real election – November, 2009 in Tacoma Park, Maryland. It was considered to be the first binding secret-ballot election for public office that used an electronic end-to-end verifiable voting system. The election was deemed successful in that there were no significant reported problems. Voters were able to verify that their votes were recorded correctly, and they were able to verify with reasonable certainty that all the votes were counted correctly (Carback et al., 2010).

2.2 Privacy (vote privacy and coercion resistance) via confirmation codes

To satisfy the privacy goals of vote privacy and coercion resistance, Scantegrity votes are recorded using randomly generated confirmation codes, one confirmation code for each candidate on a ballot (Chaum et al., 2008a). In order to distinguish between the candidates, the confirmation codes must be unique for the candidates within a particular election contest. Note the three confirmation codes on the ballot at the left of Figure 1 (Chaum et al., 2008a, p.5). The confirmation codes and associated candidate names appear on the ballots next to each other during the vote, but immediately after a voter casts his or her vote, the ballot is destroyed to maintain voter privacy.

Figure 1 An example Scantegrity ballot with one election contest. For elections with multiple contests, the ballot would have contest titles above each contest’s group of candidates. The middle picture shows what the voter sees before voting. The right picture shows what the voter sees after voting for the first candidate (see online version for colours)

To help with dispute resolution (as explained in the Individual Verifiability and Dispute Resolution section, below), ballot confirmation codes are hidden with invisible ink (Chaum et al., 2008a). Figure 1 shows a ballot in three stages – the left picture shows the ballot before confirmation codes are hidden, the middle picture shows the ballot as presented to a voter, and the right picture shows the ballot after the voter voted for Alice. The voter voted for Alice by using a special decoder pen to rub the oval adjacent to ‘Alice’. The rubbing caused Alice’s confirmation code to be revealed.

Note the torn receipt in the bottom-right corner of Figure 1. After marking his or her vote, the voter may detach and take home the ballot’s receipt. The receipt contains the ballot’s unique id (0001 in the figure) and the voter’s vote(s), in the form of a confirmation code, transcribed onto the receipt by the voter. Upcoming sections will explain how the receipts and their confirmation codes are used to satisfy the goals of
individual verifiability and universal verifiability. The confirmation codes also serve to satisfy the privacy goals. The random nature of the confirmation codes means that there’s no way for an adversary to decrypt the confirmation codes and determine who a voter voted for based on the voter’s ballot receipt. That supports the privacy goal of coercion resistance.

After the voting has finished, the voting authority publishes voters’ ballot id and confirmation code pairings on a public election website (Chaum et al., 2008a). Once again, the confirmation codes are resistant to decryption, this time from the public. Thus, the confirmation codes support the goal of vote privacy.

Private voting booths have helped to further the goal of coercion resistance by preventing potential coercers from seeing voters’ ballots. However, such coercion resistance can be compromised by a coercer’s insistence that a voter records his or her vote with a picture or video (easily taken with a cell phone). This coercion ploy can be thwarted by subjecting voters to searches prior to voting. In Post (2010) proposed a less intrusive strategy – allowing voters to re-vote. That way, potential coercers cannot be sure that voters’ pictures show a trial vote or a final vote.

2.3 Individual verifiability and dispute resolution

Publishing voters’ confirmation codes on a public election website is done in order to satisfy the goal of individual verifiability. For a particular election contest, if the voter claims that his or her receipt’s confirmation code is different from what’s on the public election website, then the dispute is resolved by privately revealing the voter’s ballot confirmation codes for the candidates in the disputed election contest (Chaum et al., 2008a). If the voter’s receipt shows a confirmation code that does not match any of the privately revealed confirmation codes, then it is assumed that the voter

• accidentally transcribed the confirmation code incorrectly
• intentionally wrote the confirmation code incorrectly in order to give the impression of a faulty, invalid election.

In both cases, the dispute would not affect the validity of the election. On the other hand, if the voter’s receipt shows a confirmation code that matches one of the privately revealed confirmation codes (but not the one on the public website), that’s an indication of

• a scanner error
• someone changing the switchboard’s flag values.

In both cases, the dispute would normally lead to an investigation (Chaum et al., 2008a).

To satisfy the goal of individual verifiability, voters must not only be able to verify that their confirmation codes were recorded correctly (as described above), but they must also be able to verify that their selected confirmation codes led to tallies for their selected candidates. That verification process will be explained in the upcoming Universal Verifiability via Randomised Partial Checking section.
2.4 Mix network

Normally, election results are made public by showing the total number of votes received for each candidate. To satisfy the goal of privacy and ensure that voters’ votes remain secret, there must be no observable link between the publicly displayed vote totals and the voters’ votes. There are several cryptographic models used to ensure that such links remain secret, and mix networks are one of the common models (Magkos et al., 2007). A mix network uses a series of servers (mixes) to permute input values to output values such that the correspondence between the inputs and outputs can be determined only with the help of the mix network’s keys, one per mix server. Figure 2, a modified version of (Jakobsson et al., 2002, p.340), illustrates how four mix servers can be used to encrypt voters’ votes prior to being tallied.

Figure 2: Four servers, $S_1$, $S_2$, $S_3$, and $S_4$, in a mix network, performing four permutations of voters’ original votes. O and M represent candidates Obama and McCain, as they might have appeared on the input and output points for the first mix server.

2.5 Switchboard

With a mix network, a series of mix servers implement a series of permutations, one permutation per mix server. On the other hand, Scantegrity uses just one server, which implements a switchboard with two permutations. Such simplicity should lead to less expensive hardware and software requirements. Also, it should make it easier for voters to understand what’s going on, and that helps to improve trust in the system (Chaum et al., 2008b).

In Figure 3 (Chaum et al., 2008a, p.5), table R implements an example switchboard that permutes table Q’s 15 cell values to the 15 cells in table S. Table R’s q-pointer and s-pointer columns form the switchboard’s two permutations. For example, the 11th q-pointer value (0001, 1) maps to the first candidate (1) on the first ballot (0001) in table Q. The 11th s-pointer value (2, 2) maps to the second column (2) in the second row (2) in table S. Note the heading for table S’s second column – Bob. Votes that connect to table S’s second column will be tallied as votes for Bob. To verify that a vote for the first candidate on the first ballot is supposed to be a vote for Bob, note that that candidate’s confirmation code is 7LH and table P indicates that ballot 0001’s confirmation code for Bob is indeed 7LH. Table P will be explained in more detail in the upcoming Ballot Auditing section.

As explained in the mixed network section above, table R’s permutations encrypt the associations between the between the voters’ votes in table Q and the public vote tallies in table S. This simple encryption scheme, with just two permutations, is resistant to
brute-force attacks because all decrypted values (decrypted values are the permutation connections between voted-for candidates and vote tallies) are equally likely. Thus, successful decryptions cannot be recognised as successful. Except for the connections revealed for audited ballots (explained shortly) and for the RPC process, the permutations are kept secret, known only to the voting authority. Such secrecy supports the goal of coercion resistance.

Figure 3 An example Scantegrity setup. Table P, hidden, stores confirmation code-candidate associations for five ballots. Table Q stores confirmation code orderings as they appear on five ballots. Table R implements a two-permutation switchboard. Table S will store tallied votes (see online version for colours)

2.6 Ballot auditing

In a Scantegrity-run election, ballot auditing takes place both before and after the actual voting (Chaum et al., 2008a). Ballot auditing is when unused ballots and their switchboard connections are revealed. Ballot auditing can help to protect against various types of attacks. One such attack would be the printing of identical confirmation code values for multiple candidates within a single election contest on a single ballot. Presumably, the repeated confirmation codes would be associated with the candidate for whom the cheating is intended to favour. Another such attack would be the printing of ballots where the confirmation code/candidate pairings were incorrect. For example, in Figure 3, if ballot 0001 used 7LH for Carl, that would be incorrect because table P associates 7LH with Bob. Presumably, by switching the confirmation codes between two candidates, the less popular candidate would garner more votes than the more popular candidate.

The aforementioned two attacks can be offset by verifying that the audited ballots match the confirmation code-candidate mappings in table P. According to Chaum et al. (2008a), if $A$ is the number of audited ballots, $B$ is the total number of ballots, and $F$ is the number of fraudulent ballots, then the probability that no errors will be detected (i.e., no errors found in the audited ballots) is defined by this equation:
For example, if \( A = 50, B = 100, \) and \( F = 10, \) then the probability that the fraudulent ballots will go undetected is 0.00059. With such a low probability, if all audited ballots are confirmed to have proper confirmation codes, then voters should have a high degree of confidence that the remaining unaudited ballots have correct confirmation codes (Chaum et al., 2008b).

A third possible attack would be changing the permutation links within the switchboard so that votes route to incorrect candidates in Figure 3’s table S. This attack can be offset by verifying that audited ballots’ confirmation codes connect to the proper candidate columns in table S. The connections must agree with the mappings shown in Figure 3’s table P. If all audited ballots are confirmed to have proper switchboard connections, then voters should have a high degree of confidence that the remaining unaudited ballots are routed correctly (Chaum et al., 2008b).

Auditing takes place at different times during the election process (Chaum et al., 2008a). Before election day, candidates or impartial observers are allowed to audit a randomly selected sample of ballots. On election day, voters are allowed to audit one unused ballot (separate from his or her cast vote’s receipt). After polling stations close, the general public is allowed to audit any remaining unused ballots.

There are several steps involved in auditing a ballot (Chaum et al., 2008a). Each audited ballot is stamped ‘Audit Ballot’, so they are not included in the actual vote. For each audited ballot, all of its confirmation codes are revealed and those confirmation codes’ mappings to specific candidates are revealed from table P. The revealed ballots’ links through the switchboard are revealed so auditors can verify that the links connect the ballots’ choices to the proper candidate columns in table S. See Figure 4, which shows the revealed information for one audited ballot from Figure 3’s Scantegrity setup.

Figure 4  An audited Scantegrity ballot
2.7 Universal verifiability via Randomised Partial Checking

While ballot auditing can assure voters that the remaining unused ballots are routed correctly, ballot auditing does not directly check votes that have been cast. On the other hand, Randomised Partial Checking (RPC) does check cast votes and thus helps to satisfy the goal of universal verifiability (Chaum et al., 2008a, 2008b; Jakobsson et al., 2002), where anyone can verify that the votes have not been modified on their way through the mix network. Specifically, RPC verifies with a specified degree of statistical certainty that individual voters’ recorded votes link to the proper candidates in the resulting vote tallies. Here’s how it works. After the polling stations close, half of the permutation links are revealed and auditors are encouraged to verify for each revealed permutation link that if a vote is cast on one side of the link, then a vote should be cast on the other side as well. Likewise, if a vote is not cast on one side of the permutation, then a vote should not be cast on the other side as well. This is best illustrated by looking at examples. Figure 5 (Chaum et al., 2008a, p.8) shows an example of information that could be revealed as part of the RPC process for the Scantegrity setup shown in Figure 3. Note how table R’s fifth q-pointer value (0001, 2) indicates a cast vote with a check mark in the flag column. That vote matches the cast vote for the second candidate in table Q’s first row, as indicated by ballot 0001’s WT9 appearing in column 2. Also in Figure 5, note how table R’s first s-pointer value (2, 1) indicates no cast vote since there is no check mark in the flag column. That no vote matches the lack of a check mark for the first candidate in table S’s second row.

Figure 5 Scantegrity’s post-election revealed information. RPC reveals half of table R’s permutations, only one revealed permutation per row in table R

Note the blank rows in Figure 5’s tables Q and R. The blank rows are for the thrown-out audited ballot and its associated permutation links. The audited ballot and its associated permutation links are shown in Figure 4.

Note that for each switchboard row, a random number generator is used to decide which one of the two permutation links is revealed. Revealing just one of the two links supports the goal of coercion resistance. If a coancer were able to view a complete connection between a voter’s confirmation-code vote and the confirmation code’s vote
tally column, and the coercer knew the voter’s confirmation code via the voter’s ballot receipt, then the coercer would know which candidate the voter voted for (Jakobsson et al., 2002).

With RPC, if any of the revealed permutation links have input and output values that do not match (i.e., one end shows a vote and one end shows no vote), that indicates an error in the voting process. This error detection mechanism is used to provide strong statistical support for the validity of a given switchboard, while at the same time not revealing full connections between values in tables Q and S (Jakobsson et al., 2002). As part of the random partial check, if half of a switchboard’s permutation links are revealed, and an adversary introduces one error (changes a yes vote to a no vote or vice versa), then there is a 50% chance that the error will be detected (Jakobsson et al., 2002). If there are n errors, then there’s a $1 - (1/2)^n$ certainty that at least one of the errors will be detected during the RPC audit procedure. When put in the context of Scantegrity’s switchboard, the n errors would equate to n incorrect flag values in either table R or table S. If there were 10 such errors, then there would be a $1 - (1/2)^{10}$ certainty of detecting at least one of the errors, which is approximately a 99.9% certainty.

If errors are detected, Jakobsson et al. suggested either
- fixing the errors and continuing
- launching an investigation to determine how the errors were introduced.

If no errors are detected, it’s possible that some errors exist, but went undetected. When RPC is used for activities where having a few errors is acceptable, then no additional checking is necessary. Typically, elections are decided by wide enough margins so that a few vote errors will not affect the outcome (Jakobsson et al., 2002).

### 2.8 Universal verifiability of tallied vote count

After the polling stations close, the Scantegrity system publishes the table of flagged votes (table S in Figure 5) on a public election website (Chaum et al., 2008a). That initiative satisfies the goal of universal verifiability. Anyone can view the table on the website and independently count the flagged votes for each candidate. By going through this process, individuals can verify that the election authority correctly computed and reported the total number of flagged votes for each candidate.

### 2.9 Inherent vulnerability of software

Although Scantegrity provides strong verifiability safeguards with its ballot auditing and RPC measures, such measures are only as good as the ability to execute them correctly. This concept applies to all software, not just Scantegrity software. Carrying the concept to its logical conclusion, Thompson (1984) (creator of the UNIX operating system) found that even if a program’s source code is correct, errors can be introduced into the program’s executable code by the program’s compiler. This lack of trust is reinforced by the well-understood notion that proving a program’s correctness is an NP-complete problem (Mercuri and Neumann, 2003b). Relating this issue to elections, Rebecca Mercuri, a leading authority on electronic elections, stated:
“Fully electronic systems do not provide any way that the voter can truly verify that the ballot cast corresponds to that being recorded, transmitted, or tabulated. Any programmer can write code that displays one thing on a screen, records something else, and prints yet another result. There is no known way to ensure that this is not happening inside of a voting system.” (Mercuri, 2007)

Putting this lack-of-software-trust issue in the context of the Scantegrity system, Scantegrity relies on commitment to ensure that certain items remain fixed after they are first determined – mappings between confirmation codes and candidate names, ballot confirmation codes, and switchboard permutation values. A commitment scheme is the concept of an entity being able to publish or transmit encrypted information at time X and reveal/decrypt the data at time Y such that it is impossible for the encrypting entity to reveal information that is different from the original information. In other words, when the information is originally published or transmitted, it is committed/fixed. Even if the commitment algorithms are sound, there’s no guarantee that the commitment algorithms are implemented and executed correctly. Also, there’s no guarantee that the vote tallying process uses the committed values properly.

A programmatic attack is an attack that involves a malicious code implementation of an otherwise sound algorithm. There are several ways to implement a programmatic attack. If the programmer is confident that no one else will see the program’s source code, then the programmer can embed the malicious behaviour into the original program code. However, programs that implement public policy, such as voting programs, would most likely have their source code inspected by a group of people. For a proprietary program, the program inspectors would be limited to a group of people within an organisation. For an open source program, the program inspectors would be the general public. In either case, proprietary or open source, to prevent detection, the malicious code must not be revealed in the original source code. The malicious code could be introduced either with the help of a specialised compiler (Thompson, 1984) or by secretly overlaying a malicious executable file on top of the original executable file on the program’s server computer. For the latter scenario, the malicious executable file would normally be created in secret by compiling a malicious version of the program’s original source code. Although no malicious intent was proven, in a 1984 school board election in Maryland, an audit showed that an incorrect version of a ballot-counting program had been used (Saltman, 1988). After the correct program was installed and run, one of the prior election results had to be reversed.

3 Methodology

The Scantegrity voting system is open source. The academic community generally considers open source code to be harder to break than secret code because open source code normally gets subjected to more scrutiny prior to its general use (Mercuri and Neumann, 2003a). However, as discussed above, even with open source, there are concerns with trusting the Scantegrity software to do what it’s supposed to do. With that in mind, I developed a scaled-down version of Scantegrity that implements the original Scantegrity application’s vote casting and vote counting functionality, but it makes no attempt to tie the functionality to voting hardware (e.g., optical scanner ballots, a results website). This study’s prototype Scantegrity application implements a programmatic attack that changes votes to a favoured candidate, so the favoured candidate wins the
The Scantegrity attack program implements a scaled-down version of Scantegrity that handles one election contest using five ballots to vote for three candidates. Initially, as shown in Figure 6, the program displays all four Scantegrity tables with all the data visible. At this point, an actual Scantegrity voting system would randomly select ballots for auditing purposes. To maintain focus on the attack, the attack program skips the pre-vote audit. This omission does not detract from the attack’s efficacy because the attack program’s algorithm handles the after-vote audit in the same manner that would be needed for a pre-vote audit.

The information in Figure 6 is available to the election authority, but it is not available to the general public. After clicking the “Allow voting to begin” button, the program displays what would be available to the general public. Figure 7 shows an example of that display. At this point, for each of the five ballots, the user can cast a vote by clicking on one of the candidate buttons.

After the user finishes voting, the user clicks one of four buttons to initiate the vote count – the “Count the votes fairly” button or one of the three “Make candidate # win” buttons. When the program detects one of those four buttons being clicked, the program not only counts the votes, it also audits any unused ballots. For each audited ballot, the
program displays all three candidate name/confirmation code pairings, as exemplified by the fourth ballot in Figure 8’s table Q. For each audited ballot, the program displays both links in the ballot’s three associated rows in table R. For example, see rows 7, 9, and 13 in Figure 8’s table R.

**Figure 7** The ScantegrityAttack program’s display just before voting takes place. To cast a vote on a ballot, the user should click one of the candidate buttons in the ballot’s row (see online version for colours)

When the user clicks the “Count the votes fairly” button, the program uses the original links in table R’s switchboard to generate check marks in tables R and S and then, as part of the RPC process, randomly decides which of the two links to reveal for each row in table R. For an example of a fair election and its revealed RPC links, see Figure 8. To verify that the 4 votes at the left are correctly routed to the 4 votes in the outcome table – 2 votes for Alice, 1 vote for Bob, and 1 vote for Carl – the q-pointer and s-pointer values in Figure 6’s table R can be examined. To perform an RPC audit like those in an
actual Scantegrity election, Table R’s revealed link values can be analysed. For example, in Figure 8, table R’s first row has a link that connects table R’s first row to table S’s second row, first column, and both ends of the link are unflagged (as indicated by the omission of check marks in table R’s row 1 and table S’s row 2, column 1). As another example, table R’s fifth row has a link that connects table R’s fifth row to table Q’s first row, second column, and both ends of the link are flagged (as indicated by the check mark in table R’s row 5 and the confirmation code in table Q’s row 2, column 1).

When the user clicks one of the “Make candidate # win” buttons, where # refers to candidate number 1, 2, or 3, the program attempts to ensure that the chosen candidate wins the election. More specifically, the program attempts to ensure that

- the chosen candidate has the most check marks in table S
- the cheating is difficult to detect.

Figure 9 shows the result after the user clicks the “Make candidate 3 win” button. Note that when compared to Figure 8’s table S, one of Alice’s votes has been moved to Carl’s column. Also note that all of the revealed links in table R match the committed links in Figure 6’s table R. Finally, note that each revealed link has matching flag values at each of its ends. By ensuring that the revealed links do not reveal any errors during the RPC and ballot auditing processes, the election authority will be disinclined to launch an investigation and/or perform a manual recount. Next, I’ll describe how the vote movement and link reveal processes are implemented.

**Figure 9** The ScantegrityAttack program’s display after “Make candidate 3 win” has been clicked. In table S, note that the third candidate, Carl has the most votes (see online version for colours)

### 3.2 Attack strategy

As described in the prior section on commitment, table R’s permutation link values are committed. That means that the original link values can be looked up later on and there is assurance that the values will be unchanged. The attack program takes this commitment into account. The program selectively overrides some of the committed permutation links in the switchboard, but makes sure that the overridden links are not revealed as part of its
Electronic voting with Scantegrity: analysis and exposing a vulnerability

RPC implementation. If the overridden links were revealed, then their values would contradict the committed links. If the contradicted committed links were publicly available, then the attack could be detected and an investigation would presumably ensue. Although the Scantegrity literature does not explicitly address the issue of making committed links publicly available for the RPC-revealed links, it is reasonable to assume that they would be made publicly available, in an attempt to detect permutation link changes.

When the user clicks one of the “Make candidate # win” buttons, the program counts the votes correctly and then enters a loop that repeatedly moves a vote to the favoured candidate until the favoured candidate has the most votes. To move a vote, the program searches for two s-pointer links in table R that can be swapped – one s-pointer value associated with a vote for a non-favoured candidate and one s-pointer value associated with a vote against the favoured candidate. Specifically, the program searches for a table R s-pointer value that (1) is part of a flagged row, (2) points to one of the non-favoured candidates in table S, and (3) can be hidden. To find such a value, the ‘can be hidden’ criterion means that the search skips table R audited rows and rows where one of the links has already been revealed as part of the RPC process. If a value is found, then the program searches for another table R s-pointer value that can be hidden, this time the value must be part of an unflagged row and must point to the favoured candidate in table S. If both values are found, the program swaps their values so that the favoured candidate gains a check mark in table S and the non-favoured candidate loses a check mark in table S. To satisfy the RPC requirement of revealing half of the links in table R and to avoid detection of the swap, the program hides the swapped s-pointer links and reveals the associated q-pointer links.

Figure 9 shows the result after the user clicks the “Make candidate 3 win” button. To support candidate 3’s victory, the attack program swapped the fourth and fifth s-pointers in table R. To enable the user to see the swapped values, the attack program provides a “Show the swapped links” button. As shown in Figure 10, when the user clicks that button, the program displays the swapped link values and changes the button’s label to “Hide the swapped links”.

**Figure 10** The ScantegrityAttack program’s display after “Show swapped links” has been clicked. Note that the “Show swapped links” button has toggled to a “Hide swapped links” button (see online version for colours)
The attack incorporates several features that are intended to reduce suspicion of a fraudulent election. The attack chooses to remove votes from candidates in a manner such that the displayed vote distribution will mimic the correct vote distribution as closely as possible. That way, the displayed outcome will be more believable. The attack does not remove a vote from a candidate if the display shows that the candidate has received just one vote. If that rule were not in place, and a candidate’s vote count were reduced to zero votes, then voters who voted for that candidate would be able to prove voter fraud. Finally, the attack attempts to reveal q-pointer and s-pointer values in a manner that mimics the flipping-a-coin mechanism prescribed by the Scantegrity algorithm. This is not as straightforward as one might think since, as described above, the mechanism for moving a vote requires hiding s-pointer links exclusively. To offset this behaviour, the attack searches for q-pointers to hide until the number of hidden q-pointer links and s-pointer links is balanced.

4 Findings

No formal experiment was conducted to determine whether this study’s attack would be detected by voters in a real election. However, when student observers were exposed to a mock Scantegrity election as shown in Figures 6–10, there was no difference in being suspicious of an attack between observers who saw a Scantegrity election with an attack and observers who saw a Scantegrity election with no attack. This finding was expected since this study’s ballot auditing and RPC mirror the verification techniques used in the real Tacoma Park, Maryland Scantegrity election described earlier.

4.1 Thwarting the attack

With many programs, by simply looking at the program’s output, it’s relatively easy to get a general sense of whether the program is doing what it’s supposed to do. For programs where that does not work, auditing code can be added which enables users to see how the program processes the input and generates the output. Such auditing can help to verify the program’s correctness.

With voting programs, looking at the program’s output does not help much in terms of gauging the program’s correctness, particularly with close elections. Auditing can help, but in a limited manner. There is an inherent tension in voting systems between needing to reveal information in order to support the goal of verifiability and needing to hide information in order to support the goal of privacy. That tension can inhibit a voting system’s ability to reveal too much information as part of an audit. The RPC process is a form of auditing in that it reveals some of the underlying information used to generate the final results. More specifically, it reveals 50% of the switchboard’s links. Unfortunately, if the programmatic attack described above is allowed to take place, that auditing would not reveal corrupt behaviour in the program.

To thwart a voting system RPC-based programmatic attack, the open source voting program’s source code needs to be compiled with a standard compiler and the resulting executable file needs to be used for the vote. To ensure that happens, the election authority should take care to store the correctly compiled executable code in a safe place. Presumably, such safety could be ensured by storing the executable file on an internet-disabled secure computer in a locked room.
4.2 Fraudulent election detection

Even if precautions are taken as described above to ensure that the Scantegrity application is not compromised, hardware errors can exist. Such hardware errors can result in incorrectly recorded votes and/or improperly connected vote tallies within the switchboard. This subsection describes how to safeguard against those problems by clarifying an issue that was not fully addressed in any of the reviewed Scantegrity literature. It describes how to verify, with a required minimum probability, that a particular Scantegrity election outcome is correct.

As stated earlier, with RPC, if there are \(n\) errors (i.e., \(n\) incorrect flag values) in a Scantegrity switchboard table, then this probability holds:

\[
\Pr[\text{at least 1 error detected when there exist } n \text{ errors}] = \left(\frac{1}{2} \right)^n.
\]

If there were a law or policy in place that required at least one error to be detected (if errors exist) with a probability of at least \(x\), then, solving for \(n\) in the above probability equation, the number of errors would need to be:

\[
n \geq \left\lceil \log_2 \left( \frac{1}{1-x} \right) \right\rceil.
\]

For example, if the minimum required probability \(x\) equalled 0.999 (which is equivalent to 99.9%), then the number of errors \(n\) would need to be 0 or at least 10.

Putting this in the context of an election, suppose that candidate Alice received 3000 votes and candidate Bob received 3100 votes. As described above, if the minimum required probability for detecting an error (if errors exist) was 0.999, then the number of errors would need to be at least 10. With 3000 votes for Alice and 3100 votes for Bob, for the election to be fraudulent (i.e., for the election outcome to be incorrect), then there would need to be 50 errors, all in favour of Alice. With 50 errors, the probability of detecting at least one of the errors would far exceed the required 0.999 value. Thus, no special effort would be required to remedy the possibility of undetected errors.

On the other hand, suppose that candidate Alice received 3048 votes and candidate Bob received 3052 votes. With those tallies, for the election to be fraudulent, there would need to be just 2 errors, both in favour of Alice. With 2 errors, the probability of detecting at least one of the errors would be \(1 - (1/2)^2 = 0.75\), which is less than the 0.999 value required in our current example. The solution is to generate multiple switchboards, each with its own randomly assigned permutations (Chaum et al., 2008b). For our current example, we would need 5 switchboards because the 2 errors would then appear in all 5 switchboards, making a total of \(5 \times 2 = 10\) errors. With 10 errors, the probability of detection would be approximately 0.999.

Keeping with our current example of requiring a probability of at least 0.999 for detecting at least one error when errors exist, the maximum number of switchboards that would be required is 10. That situation would arise if an election were decided by 1 or 2 votes. If that were the case, then just one vote error in favour of the voter with fewer votes would change the election outcome. For that one error to be detected with a 0.999 probability, it would have to be manifested in 10 different switchboards.

Jakobsson et al. (2002) opined that if there existed a greater than 50% chance that election errors would be detected, then would-be adversaries would not attempt to
introduce errors, due to their fear of getting caught. With at least 10 errors and a 99.9% chance that election errors would be detected, then would-be adversaries would be even less likely to introduce errors.

5 Conclusions

Scantegrity introduces some vulnerabilities that are different from those experienced by traditional voting systems. For example, an adversary could

- change confirmation code-candidate mappings and associated switchboard routes
- flip/change flags in an election’s switchboard
- modify the Scantegrity software so that it behaved differently than it was supposed to.

Defenses exist for all three of these attacks and they have been discussed above – ballot auditing, RPC, and open source software, respectively.

With Scantegrity being so new, specific criticisms of Scantegrity were not found in the literature. However, it’s clear that Scantegrity is not impervious to attacks. Despite the defenses mentioned above, open source software does not provide failsafe protection against programmatic attacks. Thus, as noted in the literature review (Mercuri, 2007), it is important that election authorities ensure that a correct voting program’s executable code is not replaced by executable code made from a malicious version of the voting program. This paper has corroborated that view by presenting an attack that relies on overlaying malicious Scantegrity code on top of correct Scantegrity code.

This paper is limited in that no attempt was made to tie the Scantegrity attack software to actual voting hardware. Consequently, it was not possible to fully test whether the attack’s interface would be able to fool users in a real election. It is recommended that such testing be conducted after the Scantegrity attack software is made more robust. Although not directly related to this study’s attack, it is recommended that re-voting functionality be added to Scantegrity. As noted above, re-voting functionality has the potential to address the increasing vulnerability of voting to coercion (due to the increasing availability of recording devices such as cell phones).

References


