Helios: Web-based Open-Audit Voting

Ben Adida
ben_adida@harvard.edu
Harvard University

Abstract

Voting with cryptographic auditing, sometimes called open-audit voting, has remained, for the most part, a theoretical endeavor. In spite of dozens of fascinating protocols and recent ground-breaking advances in the field, there exist only a handful of specialized implementations that few people have experienced directly. As a result, the benefits of cryptographically audited elections have remained elusive.

We present Helios, the first web-based, open-audit voting system. Helios is publicly accessible today: anyone can create and run an election, and any willing observer can audit the entire process. Helios is ideal for online software communities, local clubs, student government, and other environments where trustworthy, secret-ballot elections are required but coercion is not a serious concern. With Helios, we hope to expose many to the power of open-audit elections.

1 Introduction

Over the last 25 years, cryptographers have developed election protocols that promise a radical paradigm shift: election results can be verified entirely by public observers, all the while preserving voter secrecy. These protocols are said to provide two properties: *ballot casting assurance*, where each voter gains personal assurance that their vote was correctly captured, and *universal verifiability*, where any observer can verify that all captured votes were properly tallied. Some have used the term "open-audit elections" to indicate that anyone, even a public observer with no special role in the election, can act as auditor.

Unfortunately, there is a significant public-awareness gap: few understand that these techniques represent a fundamental improvement in how elections can be audited. Even voting experts who recognize that open-audit elections are "the way we'll all vote in the future" seem to envision a *distant* future, not one we should consider for practical purposes yet. The few implementations of open-audit elections that do exist [3, 2] have not had as much of an impact as hoped, in large part because they require special equipment and an in-person experience, thus limiting their reach.

We present Helios, a web-based open-audit voting system. Using a modern web browser, anyone can set up an election, invite voters to cast a secret ballot, compute a tally, and generate a validity proof for the entire process. Helios is deliberately simpler than most complete cryptographic voting protocols in order to focus on the central property of *public auditability*: any group can outsource its election to Helios, yet, even if Helios is fully corrupt, the integrity of the election can be verified.

Low-Coercion Elections. Voting online or by mail is typically insecure in high-stakes elections because of the coercion risk: a voter can be unduly influenced by an attacker looking over her shoulder. Some protocols [13] attempt to reduce the risk of coercion by letting voters override their coerced vote at a later (or earlier) time. In these schemes, the privacy burden is shifted from vote casting to voter registration. In other words, no matter what, *some truly private interaction* is required for coercion resistance.

With Helios, we do not attempt to solve the coercion problem. Rather, we posit that a number of settings—student government, local clubs, online groups such as open-source software communities, and others—do not suffer from nearly the same coercion risk as high-stakes government elections. Yet these groups still need voter secrecy and trustworthy election results, properties they cannot currently achieve short of an in-person, physically observable and well orchestrated election, which is often not a possibility. We produced Helios for exactly these groups with low-coercion elections.

Trust no one for integrity, trust Helios for privacy. In cryptographic voting protocols, there is an inevitable compromise: unconditional integrity, or unconditional privacy. When every component is compromised, only one of those two properties can be preserved. In this work, we hold the opinion that the more important property, the one that gets people's attention when they understand open-audit voting, is unconditional integrity: even if all election administrators are corrupt, they cannot convincingly fake a tally. With this design decision made, privacy is then ensured by recruiting enough trustees and hoping that a minimal subset of them will remain honest.

In the spirit of simplicity, and because it is difficult to explain to users how privacy derives from the acts of multiple trustees, Helios takes an interesting approach: there is only one trustee, the Helios server itself. Privacy is guaranteed only if you trust Helios. Integrity, of course, does not depend on trusting Helios: the election results can be fully audited even if all administrators – in this case the single Helios sever – is corrupt. Future versions of Helios may support multiple trustees. However, exhibiting the power of universal verifiability can be achieved with this simpler setup.

Our Contribution. In this work, we contribute the software design and an open-source, web-based implementation of Helios, as well as a running web site that anyone can use to manage their elections at `http://heliosvoting.org`. We do not claim any cryptographic novelty. Rather, our contribution is a combination of existing Web programming techniques and cryptographic voting protocols to provide the first truly accessible open-audit voting experience. We believe Helios provides a unique opportunity to educate people about the value of cryptographic auditability.

Limitations. While every major feature is functional, Helios is currently alpha software. As such, it requires Firefox 2 (or later). In addition, some aspects of the user interface, especially for administrative tasks, require significant additional polish and better user feedback on error. These issues are being actively addressed.

This Paper. In Section 2, we briefly review the Helios protocol, based on the Benaloh vote-casting approach [5] and the Sako-Kilian mixnet [16]. In Section 3, we cover some interesting techniques used to implement Helios in a modern Web browser. Section 4 covers the specifics of the Helios system and its use cases. We discuss, in Section 5, the security model, some performance metrics, and features under development. We reference related work in Section 6 and conclude in Section 7.

2 Helios Protocol

This section describes the Helios protocol, which is most closely related to Benaloh's Simple Verifiable Voting protocol [5], which itself is partially inspired by the Sako-Kilian mixnet [16]. We claim no novelty, we only mean to be precise in the steps taken by voters, administrators, and auditors, and we mean to provide enough details for an able programmer to re-implement every portion of this protocol.

2.1 Vote Preparation & Casting

The key auditability feature proposed by Benaloh's Simple Verifiable Voting is the separation of ballot preparation and casting. A ballot for an election can be viewed and filled in by anyone at any time, without authentication. The voter is authenticated only at *ballot casting* time. This openness makes for increased auditability, since *anyone*, including an auditor not eligible to vote (e.g. someone from a political organization who has already voted), can test the ballot preparation mechanism. The process is as follows between Alice, the voter, and the Ballot Preparation System (BPS):

1. Alice begins the voting process by indicating in which election she wishes to participate.
2. The BPS leads Alice through all ballot questions, recording her answers.
3. Once Alice has confirmed her choices, the BPS encrypts her choices and commits to this encryption by displaying a hash of the ciphertext.
4. Alice can now choose to *audit* this ballot. The BPS displays the ciphertext and the randomness used to create it, so that Alice can verify that the BPS had correctly encrypted her choices. If this option is selected, the BPS then prompts Alice to generate a new encryption of her choices.
5. Alternatively, Alice can choose to *seal* her ballot. The BPS discards all randomness and plaintext information, leaving only the ciphertext, ready for casting.
6. Alice is then prompted to authenticate. If successful, the encrypted vote, which the BPS committed to earlier, is recorded as Alice's vote.

Because we worry little about the possibility of coercion, Helios can be simpler than the Benaloh system that inspired it. Specifically, the BPS does not sign the ciphertext before casting, and we do not worry about Alice seeing the actual hash commitment of her encrypted vote before sealing. (The subtle reasons why these can lead to coercion are explained in [6].)

Teaching Voters about Coercion. While online-only voting is inherently coercible, few voters are aware of this issue: many US elections today are shifting to vote-by-mail without realizing the subtle but critical change in coercibility. We take the opportunity to make this issue clear with Helios by making coercion explicit: we provide a "Coerce Me!" button at ballot casting time which allows any voter to email a potential coercer the complete proof – ciphertext, randomness, and plaintext – of how they voted. This design choice does not enable new avenues for coercion; it only makes the existing coercibility more apparent. It is our hope that Helios can thus help educate voters about this critical election issue.

2.2 Bulletin Board of Votes

In all cryptographic voting protocols, a bulletin board is made publicly available. On this bulletin board, cast votes are displayed next to either a voter name or voter identification number. All subsequent data processing is also posted for the public to download and verify. A number of distributed bulletin board protocols, including consensus algorithms, have been proposed.

In Helios, we forgo complexity and opt for the simplest possible bulletin board, run by a single server. We expect auditors to check the bulletin board's integrity over time, and enough individual voters to check that their encrypted vote appears on the bulletin board. Once again, we opt for this simplification in order to focus the user on the major advantage of the system: Helios is auditable by anyone, including watchdog organizations and individual voters themselves.

2.3 Sako-Kilian/Benaloh Mixnet

In cryptographic voting protocols that wish to preserve individual ballots and potentially support write-in votes, anonymization is typically achieved by way of a *mixnet*, where trustees each shuffle and re-randomize the cast ciphertexts before jointly decrypting them. Both the shuffling and decryption of encrypted ballots are accompanied by proofs of correctness.

We use the Sako-Kilian protocol [16], the first provable mixnet based on El-Gamal re-encryption. We note that Benaloh uses a very similar technique [5]. We chose this scheme because of its simplicity and ease of explanation, even though we know of more complex protocols [15] that achieve an order of magnitude better performance for the same assurance of integrity.

El Gamal. Recall the El-Gamal encryption scheme implemented to support semantic security: a large (1024 bits) prime p is selected, such that $p = 2q + 1$ with q also prime. A generator g of the q-order subgroup of Z_p^* is selected. A secret key $x \in \mathsf{Z}_q$ is selected, and the corresponding public key $y = g^x \bmod p$ is computed. A message m in the q-order subgroup of Z_p^* is then encrypted by selecting $r \in \mathsf{Z}_q$ and computing: $c = (\alpha, \beta) = (g^r, my^r)$. Decryption is computed as $m = \alpha^{-x}\beta$.

When $m \in \mathsf{Z}_q$, meaning that it is not necessarily in the q-order subgroup of Z_p^*, a simple mapping from Z_q to the q-order subgroup of Z_p^* is used: on input m, compute $m_0 = m + 1$ and, if $m_0^q \equiv 1 \bmod p$, output m_0, otherwise output $-m_0 \bmod p$. Upon decryption, one obtains m, and the reverse mapping is achieved as follows: if $m \leq q$, set $m_0 = m$, otherwise $m_0 = -m \bmod p$, and output $m_0 - 1$. Using these techniques, we can efficiently encrypt and decrypt messages in Z_q for q a 512-bit prime. This is the natural path for message encryption, as a typical plaintext can be any string of bits up to a certain size.

Re-encryption. The El-Gamal cryptosystem offers simple re-encryption, even when using the Z_q mapping given above. Given a ciphertext $c = (\alpha, \beta)$, a ciphertext c' can be computed by selecting $s \in \mathsf{Z}_q$ and computing $c' = (g^s\alpha, y^s\beta)$. It is clear that c' and c decrypt to the same plaintext, c with randomness r and c' with randomness $r + s$.

Sako-Kilian Shuffle & Proof. In the Sako-Kilian mixnet, all inputs are El-Gamal ciphertexts. A mix server takes N inputs, re-encrypts them using re-encryption factors $\{s_i\}_{i\in[1,N]}$ and permutes them according to random permutation π_N, so that $d_i = \mathsf{Reenc}(c_{\pi(i)}, s_i)$.

To prove that it mixed its inputs correctly, a mix server produces a second, "shadow mix," as illustrated in Figure 1. The verifier then challenges the mix server to reveal the permutation and re-encryption factors for either this shadow mix or the difference between the two mixes, i.e. the shuffle that would transform the shadow mix outputs into the primary mix outputs. An honest mix server can obviously answer either challenge, while a cheating mix server can answer at most one of those questions convincingly, and is thus caught with at least 50% probability. To increase the assurance of integrity, we ask the mix server to produce a few shadow mixes. The verifier then provides the appropriate number of challenge bits, one for each shadow mix. If the mix server succeeds at responding to all challenges, then the primary mix is correct with probability $1 - 2^{-t}$, where t is the number of shadow mixes. Choosing $t = 80$ guarantees integrity with overwhelming probability.

In Helios, we need a non-interactive proof: there are many verifiers and we do not wish to perform such heavy computation for everyone who requests it. The proof protocol described above, which is Honest-Verifier Zero-Knowledge (HVZK), is thus transformed using the Fiat-

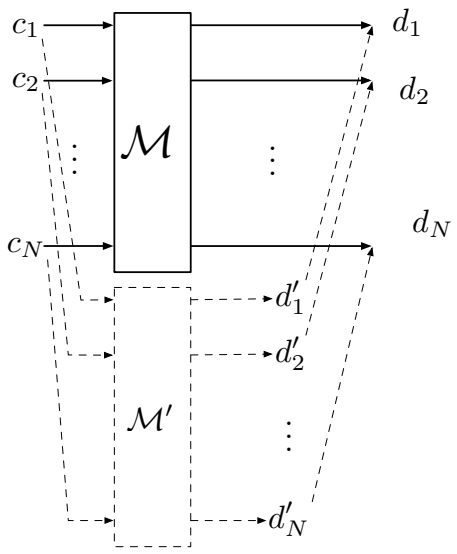

Figure 1: The "Shadow-Mix" Shuffle Proof. The mix server creates a secondary mix. If challenged with bit 0, it reveals this secondary mix. If challenged with bit 1, it reveals the "difference" between the two mixes.

Shamir heuristic [9]: the challenge bits are computed as the hash of all shadow mixes. Note how this approach is only workable if we have enough shadow mixes to provide an overwhelming probability of integrity: if there is a non-negligible probability of cheating, a cheating prover can produce many shadow mixes until it finds a set whose hash provides just the right challenge bits to cheat.

Proof of Decryption. Once an El Gamal ciphertext is decrypted, this decryption can be proven using the Chaum-Pedersen protocol [8] for proving discrete-logarithm equality. Specifically, given a ciphertext $c = (\alpha, \beta)$ and claimed plaintext m, the prover shows that $\log_g(y) = log_\alpha(\beta/m)$:

- The prover selects $w \in \mathsf{Z}_q$ and sends $A = g^w, B = \alpha^w$ to the verifier.
- The verifier challenges with $c \in \mathsf{Z}_q$.
- The prover responds with $t = w + xc$.
- The verifier checks that $g^t = Ay^c$ and $\alpha^t = B(\beta/m)^c$.

It is clear that, given c and t, A and B can be easily computed, thus providing for simulated transcripts of such proofs indicating Honest-Verifer Zero-Knowledge. It is also clear that, if one could rewind the protocol and obtains prover responses for two challenge values against the same A and B, the value of x would be easily solvable, thus indicating that this is a proof of knowledge of the discrete log and that $\log_g(y) = log_\alpha(\beta/m)$.

As this protocol is HVZK with overwhelming probability of catching a cheating prover, it can be transformed safely into non-interactive form using the Fiat-Shamir heuristic. We do exactly this in Helios to provide for non-interactive proofs of decryption that can be posted publicly and re-distributed by observers.

2.4 The Whole Process

The entire Helios protocol thus unfolds as follows:

1. Alice prepares and audits as many ballots as she wishes, ensuring that all of the audited ballots are consistent. When she is satisfied, Alice casts an encrypted ballot, which requires her to authenticate.
2. The Helios bulletin board posts Alice's name and encrypted ballot. Anyone, including Alice, can check the bulletin board and find her encrypted vote posted.
3. When the election closes, Helios shuffles all encrypted ballots and produces a non-interactive proof of correct shuffling, correct with overwhelming probability.
4. After a reasonable complaint period to let auditors check the shuffling, Helios decrypts all shuffled ballots, provides a decryption proof for each, and performs a tally.
5. An auditor can download the entire election data and verify the shuffle, decryptions, and tally.

If an election is made up of more than one race, then each race is treated as a separate election: each with its own bulletin board, its own independent shuffle and shuffle proof, and its own decryptions. This serves to limit the possibility of re-identifying voters given long ballots where any given set of answers may be unique in the set of cast ballots.

3 Web Components

We have clearly stated that Helios values integrity first, and voter privacy second. That said, Helios still takes great care to ensure voter privacy, using a combination of modern Web programming techniques. Once the ballot is loaded into the browser, all candidate selections are recorded within the browser's memory, without any further network calls until the ballot is encrypted and the plaintext is discarded. In this section, we cover the Web components we use to accomplish this goal.

3.1 Single-Page Web Application

A number of Web applications today are called "single-page applications" in that the page context and its URL never change. Gmail [10] is a prime example: clicks cause background actions rather than full-page loads. The technique behind this type of Web application is the use of JavaScript to handle user clicks:

```
<a onclick="do_stuff()" href="#">Do Stuff</a>
```

When a user clicks the "Do Stuff" link, no new page is loaded. Instead, the JavaScript function `do_stuff()` is invoked. This function may make network requests and update the page's HTML, but, importantly, the page context, including its JavaScript scope, is preserved.

For our purposes, the key point is that, if all necessary data is pre-loaded, the `do_stuff()` function may not need to make any network calls. It can update some of its scope, read some of its pre-loaded data, and update the rendered HTML user interface accordingly. This is precisely the approach we use for our ballot preparation system: the browser loads all election parameters, then leads the voter through the ballot without making any additional network requests.

The jQuery JavaScript Library. Because we expect auditors to take a close look at our browser-based JavaScript code, it is of crucial importance to make this code as concise and legible as possible. For this purpose, we use the jQuery JavaScript library, which provides flexible constructs for accessing and updating portions of the HTML Document Object Model (DOM) tree, manipulating JavaScript data structures, and making asynchronous network requests (i.e. AJAX). An auditor is then free to compare the hash of the jQuery library we distribute with that of the official distribution from the jQuery web site.

JavaScript-based Templating. Also important to the clarity of our browser-based code is the level of intermixing of logic and presentation: when all logic is implemented in JavaScript, it is tempting to intermix small bits of HTML, which makes for code that is particularly difficult to follow. Instead, we use the jQuery JavaScript Templating library. Then, we can bind a template to a portion of the page as follows:

```
$("#main").setTemplateURL(
  "/templates/election.html"
);
```

which connects to an HTML template with variable placeholders:

```
<p>
  The election hash is {$T.election.hash}.
</p>
```

The code can, at a later point, render this template with a parameter and *without* any additional network access:

```
$("#main").processTemplate(
  {'election': election_object}
);
```

3.2 Cryptography in the Browser with LiveConnect

JavaScript is a complete programming language in which it is possible to build a multi-precision integer library. Unfortunately, JavaScript performance for such computationally intensive operations is poor. Thankfully, it is possible in modern browsers to access the browser's Java Virtual Machine from JavaScript using a technology called LiveConnect. This is particularly straightforward in Firefox, where one can write the following JavaScript code:

```
var a = new java.math.BigInteger(42);
document.write(a.toString());
```

and then, from JavaScript still, invoke all of Java's `BigInteger` methods directly on the object. Modular exponentiation is a single call, `modPow()`, and ElGamal encryption runs fast enough that it is close to imperceptible to the average user. LiveConnect is slightly more complicated to implement in Internet Explorer and Safari, though it can be done [18].

3.3 Additional Tricks

Data URIs. At times in the Helios protocol, we need to produce a printable receipt when the plaintext vote has not yet been cleared from memory. In order to open a new window ready for printing without network access, we use data URIs [14], URIs that contain information without requiring a network fetch:

```
<a target="_new"
   href="data:text/plain,Your%20Receipt...">
     receipt
</a>
```

Dynamic Windows. When data URIs are not available (e.g. Internet Explorer), we can open a new window using JavaScript, set its MIME type to `text/plain`, and dynamically write its content from the calling frame.

```
var receipt = window.open();
receipt.document.open("text/plain");
receipt.document.write(content);
receipt.document.close();
```

In Safari and Firefox, this approach yields a new window in a slightly broken state: the contents cannot be saved to disk. However, in Internet Explorer, the only browser that does not support Data URIs, the dynamic window creation works as expected. Thus, in Firefox and Safari, Helios uses Data URIs, and in Internet Explorer it uses dynamic windows.

JSON. As we expect that auditors will want to download election, voter, and bulletin board data for processing and verifying, we need a data format that is easy to parse in most programming languages, including JavaScript. XML is one possibility, but we found that JavaScript Object Notation (JSON) is easier to handle with far less parsing code. JSON allows for data representation using JavaScript lists and associative arrays. For example, a list of voters and their encrypted votes can be represented as:

```
[
  {'name' : 'Alice', 'vote' : '23423....'},
  {'name' : 'Bob', 'vote' : '823848....'},
  ...
]
```

Libraries exist in all major programming languages for parsing and generating this data format. In particular, the format maps directly to arrays and objects in JavaScript, lists and dictionaries in Python, lists and hashes in Ruby.

4 Helios System Description

We are now ready to discuss the details of the Helios system. We begin with a description of the back-end server architecture. We then consider the four use cases: creating an election, voting, tallying, and auditing.

4.1 Server Architecture

The Helios back-end is a Web application written in the Python programming language [17], running inside the CherryPy 3.0 application server, with a Lighttpd web server. All data is stored in a PostgreSQL database. All server-side logic is implemented in Python, with HTML templates rendered using the Cheetah Templating engine. Many back-end API calls return JSON data structures using the Simplejson library, and the voting booth server-side template is, in fact, a single-page web applications including JavaScript logic and jTemplate HTML/JavaScript templates.

Application Software. We use the Python Cryptography Toolkit for number theory utilities such as prime number and random number generation. We implemented our own version of El-Gamal in Python, given our specific need for re-encryption, which is typically not supported in cryptographic libraries. We note that improved performance could likely be gained from optimizing our first-pass implementation.

Server Hardware. We host an alpha version of the Helios software at `http://heliosvoting.org`. The server behind that URL is a virtual Ubuntu Linux server operated by SliceHost. For the tests performed in Section 5.3, we used a small virtual host with 256 megabytes of RAM and only a fraction of a Xeon processor, at a cost of $20/month. A larger virtual host would surely provide better performance, but we wish to show the practicality of Helios even with modest resources.

4.2 Creating an Election

Only registered Helios users can create elections. Registration is handled like most typical web sites:

- a user enters an email address, a name, and a desired password.
- an email with an embedded confirmation link is sent to the given email address.
- the user clicks on the confirmation link to activate his account.

A registered user then creates an election with an election name, a date and time when voting is expected to begin, and a date and time when voting is expected to end. Upon creation, Helios generates and stores a new El-Gamal keypair for the election. Only the public key is available to the registered user: Helios keeps the private key secret. The user who created the election is considered the *administrator*.

Setting up the Ballot. The election is then in "build mode," where the ballot can be prepared, reviewed, and tweaked by the administrative user, as shown in Figure 2. The user can log back in over multiple days to adjust any aspect of the ballot.

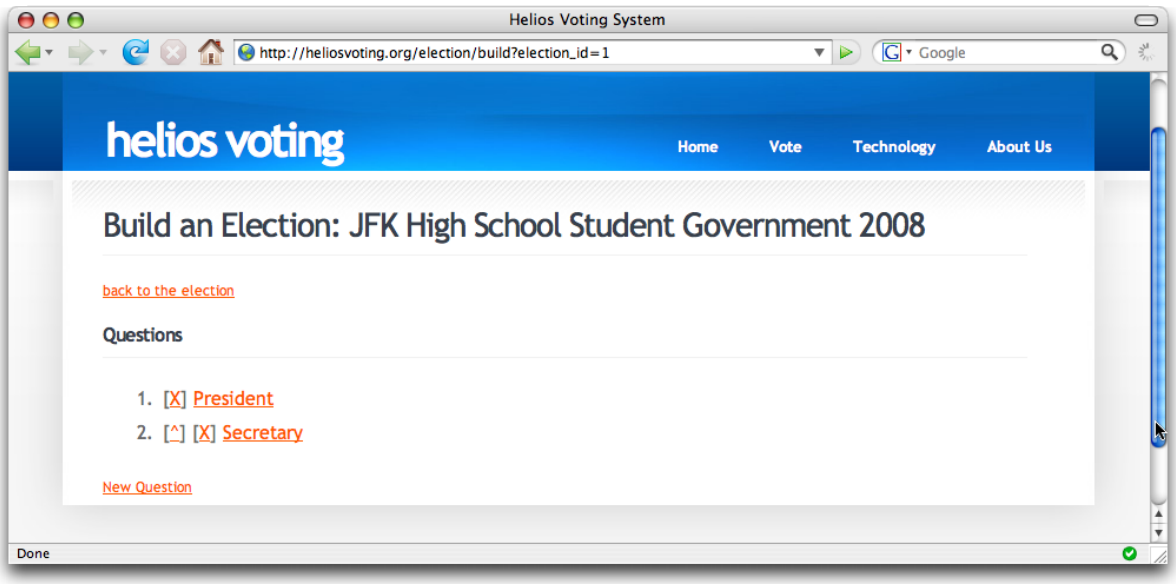

Figure 2: The Helios Election Builder lets an administrative user create and edit ballot questions in a simple web-based interface. The administrative user can log out and back in at any time to update the election.

Managing Voters. The administrative user can add, update, and remove voters at will, as shown in Figure 3. A voter is identified by a name and an email address, and is specific to a given election. Helios generates a random 10-character password automatically for each voter. At any time, the administrator can email voters using the Helios administrative interface. These emails will automatically contain the voter's password, though the administrator will not see this password at any time.

Freezing the Election. When ready, the administrative user *freezes the election*, at which point the voter list, the election start and end dates, and the ballot details become immutable and available for download in JSON form. The administrative user receives an email from Helios with the SHA1 hash of this JSON object. The election is ready for voters to cast ballots. The administrative user will typically email voters using the Helios administrative interface to let them know that the polls are open.

4.3 Voting

Alice, a voter in a Helios election, receives an email letting her know that the polls are open. This email contains her username (i.e. her email address), her election-specific password, the SHA1 hash of the election parameters, and the URL that directs her to the Helios voting booth, as illustrated in Figure 4. It is important to note that this URL does not contain any identifying information: it only identifies the election, as per the vote-casting protocol in Section 2.1.

The Voting Booth. When Alice follows the voting booth URL, Helios responds with a single-page web application. This application, now running in Alice's browser, displays a "loading..." message while it downloads the election parameters and templates, including the El-Gamal public key and questions. The page then displays the election hash prominently, and indicates that no further network connections will be made until Alice submits her encrypted ballot. (Alice can set her browser to "offline" mode to enforce this.) Every transition is then handled by a local JavaScript function call and its associated templates. Importantly, the JavaScript code can decide precisely what state to maintain and what state to discard: the "back" button is not relevant. This is illustrated in Figure 5.

Filling in the Ballot. Alice can then fill in the ballot, selecting the checkbox by each desired candidate name, using the "next" and "previous" buttons to navigate between questions. Each click is handled by JavaScript code which records Alice's choices in the local JavaScript scope. If Alice tries to close her browser or navigate to a different URL, she receives a warning that her ballot will be cleared.

Sealing. After Alice has reviewed her options, she can choose to "seal" her ballot, which triggers the JavaScript code to encrypt her selection with computationally intensive operations performed via LiveConnect. The SHA1 hash of the resulting ciphertext is then displayed, as shown in Figure 6.

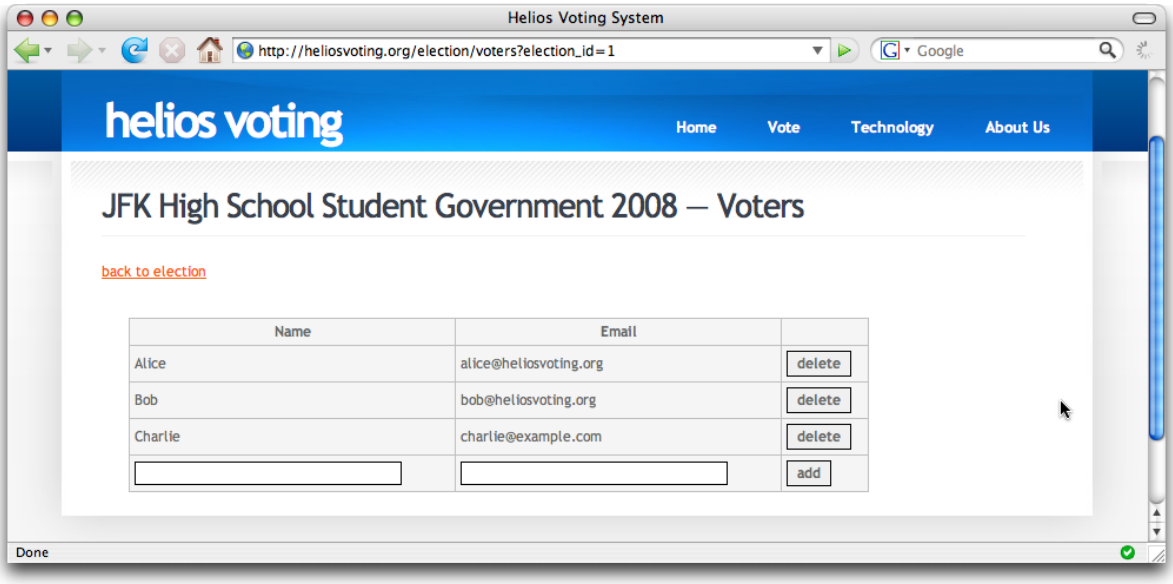

Figure 3: The Helios voter management interface.

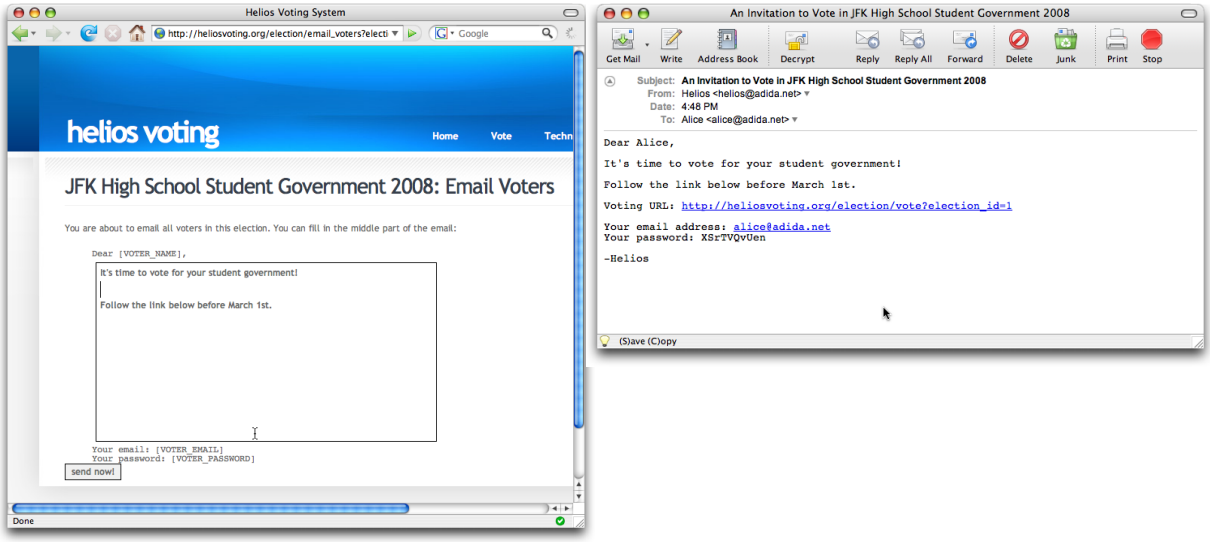

Figure 4: The administrative user can send emails to all voters. Each voter receives her password, which the administrative user does not see.

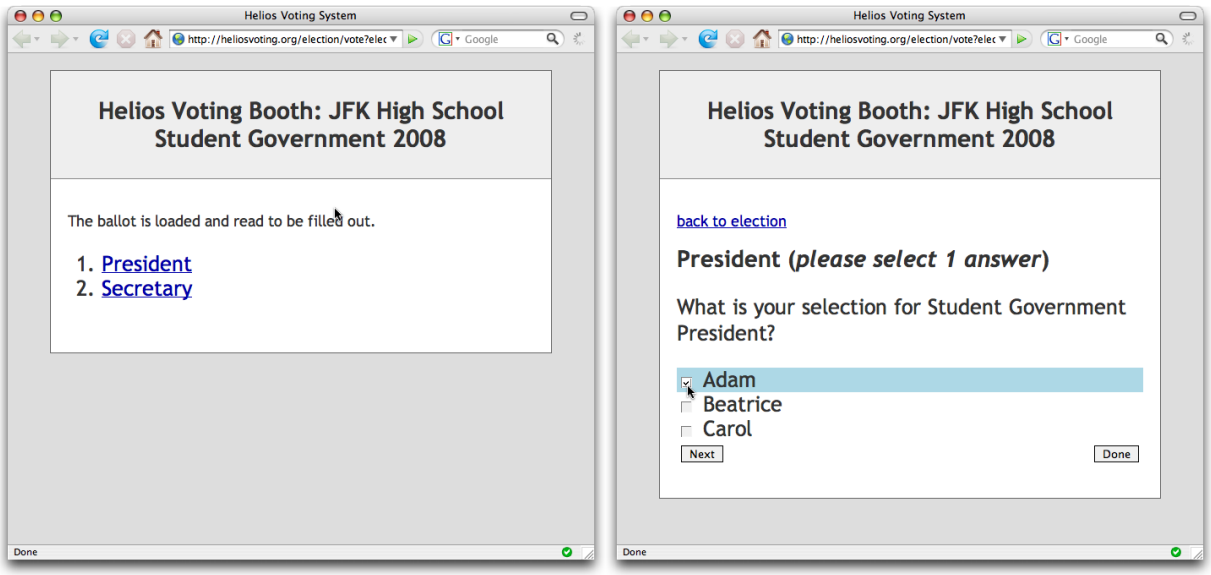

Figure 5: The Helios Voting Booth.

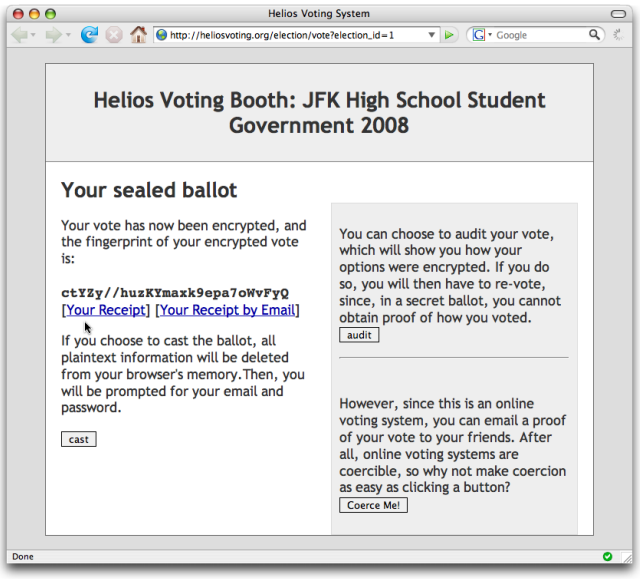

Figure 6: Sealing a Helios ballot.

Auditing. Alice can opt to audit her ballot with the "Audit" button, in which case the JavaScript code reveals the randomness used in encrypting Alice's choices. Alice can save this data to disk and run her own code to ensure the encryption was correct, or she can use the Python Ballot Encryption Verification (BEV) program provided by Helios.

Once Alice chooses to audit her ballot and the auditing information is rendered, the JavaScript code clears its encrypted ballot data structures and returns Alice to the confirmation screen, where she can either update her choices or choose to seal her options again with different randomness and thus a different ciphertext.

Casting. If Alice chooses instead to cast her ballot, the JavaScript code clears the plaintext and randomness from its scope, and presents Alice with a login prompt for her email address and password. (If Alice had set her browser to "offline" mode, she should bring it back online now that all plaintext information is cleared.) When Alice submits her login information, the JavaScript code intercepts the form submission and submits the email, password, and encrypted vote in a background call, so that any errors, e.g. a mistyped password, can be reported without clearing the JavaScript scope and thus the encrypted ballot. When a success code is returned by the Helios server, the JavaScript code can clear its entire scope and display a success message. On the server side, Helios emails Alice with a confirmation of her encrypted vote, including its SHA1 hash.

Coerce Me! As explained in Section 2, Helios provides a "Coerce Me!" button to make it clear that online voting is inherently coercible. This button appears after ballot sealing, next to the "audit" and "cast" options. When clicked, Helios opens up a new window with a `mailto:` URL that triggers Alice's email client to open a composition window containing the entire ballot information, including plaintext and randomness that prove how the ciphertext was formed. Unlike the "Audit" step, which forces Alice to create a new ciphertext, "Coerce Me!" allows Alice to continue and cast that very same encrypted vote for which she obtained proof of encryption. The distinction between these two steps highlights the difference between a coercion-free auditing process that could potentially be used with in-person voting, and the inherent coercibility of online-only voting which is made more explicit with the "Coerce Me!" button.

4.4 Anonymization

Once the voting period ends, Helios enables the anonymization, decryption, and proof features for the administrative user. Selecting "shuffle" will begin the re-encryption and permutation process. Then, selecting "shuffle proof" will trigger the mixnet proof with 80 shadow mixes. The administrative user can then opt for "decrypt", which will decrypt the shuffled ciphertexts, and "decrypt proof", which will generate proofs for each such decryption. Finally, the administrative user can select "tally" to count up the decrypted votes.

All of these operations are performed on the server side, in Python code. The results are stored in the database and made available for download in JSON form. Once all proofs are generated and the result is tallied, the server deletes the permutation, randomness, and secret key for that election. All that is left is the encrypted votes, their shuffling, the resulting decryptions, and the publicly verifiable proofs of integrity. The entire election can still be verified, though no further proofs can be generated.

4.5 Auditing

Helios provides two verification programs, one for verifying a single encrypted vote produced by the ballot preparation system with the "audit" option selected, and another for verifying the shuffling, decryption, and tallying of an entire election. Both programs are written in Python using the Simplejson library for JSON processing, but otherwise only raw Python operations.

Verifying a Single Vote. The Ballot Encryption Verification program takes as input the JSON data structure returned by the voting booth audit process. This data structure contains a plaintext ballot, its ciphertext, the randomness used to encrypt it, and the election ID. The program downloads the election parameters based on the election ID and outputs:

- the hash of the election, which the voter can check against that displayed by the voting booth,
- the hash of the ciphertext, which the voter can check against the receipt she obtained before requesting an audit,
- the verified plaintext of the ballot.

Verifying an Election. The Election Tallying Verification program takes, as input, an election ID. It downloads the election parameters, the bulletin board of cast votes, shuffled votes, shuffle proofs, decrypted votes, and decryption proofs. The verification program checks all proofs, then re-performs the tally based on the decryptions. It eventually outputs the list of voters and their respective encrypted ballot hashes, plus the verified tally. This information can be reposted by the auditor, so that

if enough auditors check and re-publish the cast ballot hashes and tally, participants can be confident that their vote was correctly captured, and that the tally was correctly performed.

5 Discussion

Helios is simpler than most cryptographic voting protocols because it focuses on proving integrity. As a compromise, Helios makes weaker guarantees of privacy. In this section, we review in greater detail the type of election for which we expect this compromise to be appropriate, as well as the security model, performance metrics, and future extensions we can make to improve Helios on both fronts.

5.1 The Need for Verifying Elections with Low Coercion Risk

It is legitimate to question whether there truly exist elections that require the high levels of verifiability afforded by cryptography, while eschewing coercion-resistance altogether. In fact, we believe that, for a number of online communities that rarely or never meet in the same physical place:

1. coercion-resistance is futile from the start, given the remote nature of the voting process, and
2. cryptographic end-to-end verifiability is the *only* viable means of ensuring *any* level of integrity.

Specifically, with respect to the auditing argument, how could a community member remotely verify anything at all pertaining to the integrity of an election process? Open-source software is insufficient: the voter doesn't know *which* software is actually running on the election server, short of deploying hardware-rooted attestation. Physical observation of a chain-of-custody process is already ruled out by the online-only nature of the community. Cryptographic verifiability, though it seems stronger than absolutely necessary, is the only viable option when only the public inputs and outputs—never the "guts"—of the voting process can be truly observed. Cryptographic auditing may be a big hammer, but it is the only hammer.

For the same reason, we believe the pedagogical value of a system like Helios is particularly strong. The contrast between classic and open-audit elections is particularly apparent in this online setting. With Helios, the voter's ability is transformed, from entirely powerless and forced to trust a central system, to empowered with the ability to ensure that one's vote was correctly captured and tallied, without trusting anyone.

5.2 Security Model & Threats

We accept the risk that, if someone compromises the Helios server before the end of an election, the secrecy of individual ballots may be compromised. On the other hand, we claim that, assuming enough auditors, even a fully corrupted Helios cannot cheat the election result without a high chance of getting caught. We now explore various attacks and how we expect them to be handled.

Incorrect Shuffling or Decryption. A corrupt Helios server may attempt to shuffle votes incorrectly or decrypt shuffled votes incorrectly. Given the overwhelming probability of catching these types of attacks via cryptographic verification, it takes only one auditor to detect this kind of tampering.

Changing a Ballot or Impersonating a Voter. A corrupt Helios may substitute a new ciphertext for a voter, replacing his cast vote or injecting a vote when a voter doesn't cast one in the first place. Even if the ballot submission server is eventually hosted separately and distributed among trustees, a corrupt Helios server knows the username and password for all users, and can thus easily authenticate and cast a ballot on behalf of a user. In this case, all of the shuffling and decryption verifications will succeed, because the corruption occurs before the encryption step.

In the current implementation of Helios, we hope to counter these attacks through extensive auditing. Previous analyses [7] have shown that it takes only a small random sample of voters who verify their vote to defeat this kind of attack. To encourage voters to audit their votes, we created the Election Tallying Verification program, available in well commented source form. The Election Tallying Verification program outputs a copy of all cast ballots, so that auditors can post this information independently. We expect multiple auditors to follow this route and re-publish the complete list of encrypted ballots along with their re-computed election outcome. This auditing may include re-contacting individual voters and asking them to verify the hash of their cast encrypted ballot. We also expect that a large majority of voters, maybe all voters, in fact, will answer at least one auditor who prompts them to verify their cast encrypted vote.

Corrupting the Ballot. A corrupt Helios may present a corrupt ballot to Alice, making her believe that she's selecting one candidate when actually she is voting for another. This kind of attack would defeat the hashed-vote bulletin-board verification, even with multiple auditors, since Alice receives an entirely incorrect receipt during the ballot casting process. Helios mitigates this risk by authenticating users only *after* the ballot has been filled

out, so users cannot be individually targeted with corrupt ballots as easily. However, a corrupt Helios may authenticate voters first (voters may not notice), or use other information (e.g. IP address) to identify voters and target certain victims for ballot corruption.

To counter this attack, we provide the Ballot Encryption Verification program, again in source form for auditors to verify. This program can be run by individual voters when they choose to audit a handful of votes before they choose to truly cast one. Alternatively, auditors, even auditors who are not eligible to vote in the election, can prepare ballots and audit them at will.

Auditing is Crucial. It should be clear from these descriptions that Helios counters attacks through the power of auditing. In addition to the raw tally, Helios publishes a list of voter names and corresponding encrypted votes. Helios then provides supporting evidence for the tally, given the cast encrypted votes, in the form of a mixnet-and-decryption proof. Verification programs are available in source form for anyone to review the integrity of the results.

However, only the individual voters can check the validity of the cast encrypted ballots. It is expected that multiple auditors will check the proof and, when satisfied, republish the tally *and* the list of cast encrypted ballots, where voters can check that their ballot was correctly recorded. Helios ensures that, if a large majority of voters verifies their vote, then the outcome is correct. However, if voters do not verify their cast ballot, Helios does not provide any verification beyond classic voting systems.

These expectations are somewhat tautological: voter-verified elections function only when at least some fraction of the voters are willing to participate in the verification process made available to them. Elections can be made *verifiable*, but only voters can actually verify that their *secret* ballot was correctly recorded.

5.3 Performance

For all performance measurements, we used the server hardware described in the previous section, and, on the client side, a 2.2Ghz Macintosh laptop running Firefox 2 over a home broadband connection. We note that performance of Firefox 2 was greatly increased when running on virtualized Linux on the same laptop, indicating that our measurements are likely a worst-case scenario given platform-specific performance peculiarities of Firefox.

Java Virtual Machine Startup. The Java Virtual Machine requires startup time. Our rough measurements indicate anywhere between 500ms and 1.5s on our client machine. During this time, the browser appears to freeze and user input is suspended. To an uninformed user, this is a usability impediment which will require further user testing. That said, it is a behavior we can easily warn users about before starting up the Ballot Preparation System, and because this happens only once per user session – not once per ballot – it is not too onerous.

Timing Measurements. We experimented with a 2-question election and 500 voters. All timings were performed a sufficient number of times to obtain a stable average mostly free of testing noise. Note that time measurements that pertain to a set of ballots are expected to scale linearly with the number ballots and the number of questions in the election. Our results are presented in Figure 7.

Operation	Time
Ballot Encryption, in browser $\|p\| = 1024$ bits	300ms
Shuffling, on server	133 s
Shuffle Proof, on server	3 hours
Decryption, on server	71 s
Decryption Proof, on server	210 s
Complete Audit, on client	4 hours

Figure 7: Timing Measurements

The Big Picture. It takes only a few minutes of computation to obtain results for a 500-voter election. The shuffle proof and verification steps require a few hours, and are thus, by far, the most computation-intensive portions of the process. We note that both of these steps are highly parallelizable and thus could be significantly accelerated with additional hardware.

5.4 Extensions

There are many future directions for Helios.

Support for Other Types of Election. Helios currently supports only simple elections where Alice selects 1 or more out of the proposed candidates. Adding write-ins and rank-based voting, as well as the associated tallying mechanisms, could prove useful. Helios may also eventually offer homomorphic-based tabulation, as they are often easier to explain and verify, though they would made greater demands of browser-based cryptography.

Browser-Based Verification. The current verification process for the ballot encryption step is a bit tedious, requiring the use of a browser and a Python program. We could write a JavaScript-only verification program that

could be provided directly by auditors while running entirely in the voter's browser to check that Helios is delivering authentic ballots. There are some issues to deal with, notably cross-domain requests, but it does seem possible and desirable to accomplish browser-only ballot encryption verification.

Similarly, it is certainly possible to audit an entire election using JavaScript and LiveConnect for computationally intensive operations. Letting auditors deliver the source code for these verification programs would allow any voter to audit the entire process straight from their browser.

Distributing the Shuffling and Decryption. For improved privacy guarantees, Helios can be extended to support shuffling and decryption by multiple trustees. The Helios server would then only focus on providing the bulletin board and voting booth functionality. Trustees would be provided with standalone Python programs that perform threshold key generation, partial shuffling and threshold decryption. They could individually audit the program's source code. With these extensions, Helios would resemble classic cryptographic voting protocols more closely, and would provide stronger privacy guarantees.

Improving Authentication. Currently, our protocol requires that most voters audit their cast ballot, otherwise the Helios server could impersonate voters and change the election outcome. Future version of Helios should consider offloading authentication to a separate authentication service. If feasible with browser-based cryptography, Helios should use digital signatures to authenticate each ballot in a publicly verifiable manner.

6 Related Work

There is a plethora of theoretical cryptographic voting work reviewed and cited in [11, 4]. We do not attempt to re-document this significant body of work here.

Open-audit voting implementations. There are only a small handful of notable open-audit voting implementations. VoteHere's advanced protocols for mixnets and coercion-free ballot casting [3] have been implemented and deployed in test environments. The Punchscan voting system [2] has also been implemented and used in a handful of real student government elections, with video evidence available for all to see.

Browser-based cryptography. Cryptographic constructs have been implemented in browser-side code in many different settings. In the late 1990s, Hushmail began providing web-based encrypted email using a Java applet. A couple of years later, George Danezis showed how to use LiveConnect for fast JavaScript-based cryptography, and the EVOX voting project [12] used similar technology to encrypt votes in a blind-signature-based scheme. The Stanford SRP project [18] also uses LiveConnect for browser-based cryptography and indicates how one can get LiveConnect to work in browsers other than Firefox. The recent Clipperz Crypto Library [1] provides web-based cryptography in pure JavaScript, including a multi-precision integer library.

7 Conclusion

Helios is the first publicly available implementation of a web-based open-audit voting system. It fills an interesting niche: elections for small clubs, online communities, and student governments need trustworthy elections without the significant overhead of coercion-freeness. We hope that Helios can be a useful educational resource for open-audit voting by providing a valuable service – outsourced, verifiable online elections – that could not be achieved without the paradigm-shifting contributions of cryptographic verifiability.

8 Acknowledgements

The author would like to thank Ronald L. Rivest, Shai Halevi, Chris Peikert, Susan Hohenberger, Alon Rosen, Steve Weis, and Greg Morrisett for alpha testing the Helios system, Adam Barth for pointing out the dynamic window creation technique for the Internet Explorer work-around, and the Usenix Security reviewers for insightful suggestions in improving and crystalizing the presentation of this work.

References

[1] Clipperz. `http://clipperz.org`, last viewed on January 30th, 2008.

[2] PunchScan. `http://punchscan.org`, last viewed on January 30th, 2008.

[3] VoteHere. `http://votehere.com`, last viewed on January 30th, 2008.

[4] Ben Adida. *Advances in Cryptographic Voting Systems*. PhD thesis, August 2006. `http://ben.adida.net/research/phd-thesis.pdf`.

[5] Josh Benaloh. Simple Verifiable Elections. In *EVT '06, Proceedings of the First Usenix/ACCURATE*

Electronic Voting Technology Workshop, August 1st 2006, Vancouver, BC,Canada., 2006. Available online at http://www.usenix.org/events/evt06/tech/.

[6] Josh Benaloh. Ballot Casting Assurance via Voter-Initiated Poll Station Auditing. In *EVT '07, Proceedings of the Second Usenix/ACCURATE Electronic Voting Technology Workshop, August 6th 2007, Boston, MA, USA.*, 2007. Available online at http://www.usenix.org/events/evt07/tech/.

[7] C. Andrew Neff. Election Confidence. http://www.votehere.com/papers/ElectionConfidence.pdf, last viewed on January 30th, 2008.

[8] David Chaum and Torben P. Pedersen. Wallet databases with observers. In Ernest F. Brickell, editor, *CRYPTO*, volume 740 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 1992.

[9] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *CRYPTO*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194. Springer, 1986.

[10] Google. Gmail. http://gmail.com.

[11] Dimitris Gritzalis, editor. *Secure Electronic Voting*. Kluwer Academic Publishers, 2002.

[12] Mark Herschberg. Secure electronic voting over the world wide web. Master's thesis, May 1997. http://groups.csail.mit.edu/cis/voting/herschberg-thesis/.

[13] Ari Juels, Dario Catalano, and Markus Jakobsson. Coercion-resistant electronic elections. In Vijay Atluri, Sabrina De Capitani di Vimercati, and Roger Dingledine, editors, *WPES*, pages 61–70. ACM, 2005.

[14] L. Masinter. The Data URL Scheme. http://tools.ietf.org/html/rfc2397, last viewed on January 30th, 2008.

[15] C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *CCS 2001, Proceedings of the 8th ACM Conference on Computer and Communications Security. November 6-8, 2001, Philadelphia, Pennsylvania, USA.*, pages 116–125. ACM, 2001.

[16] Kazue Sako and Joe Kilian. Receipt-free mix-type voting scheme - a practical solution to the implementation of a voting booth. In *EUROCRYPT*, pages 393–403, 1995.

[17] Guido van Rossum. The Python Programming Language. http://python.org, last viewed on January 30th, 2008.

[18] Thomas D. Wu. The secure remote password protocol. In *NDSS*. The Internet Society, 1998. http://srp.stanford.edu/, last visited on January 30th, 2008.