Electing a University President using Open-Audit Voting: Analysis of real-world use of Helios

Ben Adida
Harvard University
ben_adida@harvard.edu

Olivier de Marneffe
Université catholique de Louvain
olivier.demarneffe@uclouvain.be

Olivier Pereira
Université catholique de Louvain
olivier.pereira@uclouvain.be

Jean-Jacques Quisquater
Université catholique de Louvain
jjq@uclouvain.be

June 25, 2009

Abstract

In March 2009, the Université catholique de Louvain elected its President using a custom deployment of the Helios web-based open-audit voting system. Out of 25,000 potential voters, 5000 registered, and almost 4000 voted in each round of the election. The precision of the voting system turned out to be crucial: in the first round, the leader came short of winning the election by only 2 votes.

In this work, we document the new version of Helios used in this election, the specifics of the UCL deployment, and the lessons learned in this deployment. We offer suggestions on running future open-audit elections. We note at least one interesting conclusion: while it is often assumed that open-audit voting will lead to more complaints and potentially a denial-of-service attack on the auditing process, we found that, instead, complaints are likely to be more easily handled in open-audit elections because evidence and counter-evidence can be presented.

1 Background and 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* [3], 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, though some important test elections have been held in recent years, notably using the Scantegrity [7] and Prêt-à-Voter [8] systems, none have had significant stakes yet. As a result, public awareness of open-audit elections remains unfortunately low. Even voting experts who recognize that open-audit elections are "the way we'll all vote in the future" seem to envision a *very distant* future, not one we should consider for practical purposes yet.

The UCL Opportunity. In 2008, the Université catholique de Louvain (UCL) in Louvain-la-Neuve, Belgium, decided to open up its University President selection process to all members of the University, including faculty, students, and staff, via a secret-ballot election conducted through the Internet. With 25,000 eligible voters participating in an election without precedent, there was both a challenge and an opportunity. The challenge lay in providing a trustworthy result of an election with such high stakes. The opportunity to try something new was motivated by two important factors: (i) the UCL's cryptography department was consulted and given the opportunity to run the election, and (ii) the absence of precedent combined with a general distrust of typical computer-based voting made a novel, publicly verifiable approach particularly appealing.

Helios. Helios, [2], one of the first web-based open-audit voting system, was released in 2008. Given the simplicity and universality of the web, Helios was immediately a clear candidate for the UCL election. However, Helios was not ready for immediate deployment at UCL, because:

- Helios 1.0 was based on mixnets, which could not be used in an election in which votes receive different weights according to the voter's category, while election results per category need to remain unknown.
- Helios 1.0 required Google App Engine to run, which, though not a problem in theory, posed a number of problems for integration with a European University given European privacy laws.
- Helios 1.0 verifiability was implemented as a proof-of-concept, and was not scalable to rapid tallying of tens of thousands of votes, nor designed to help solving potential voter complaints.
- Helios 1.0 only implemented single-key decryption, meaning that one party would have to be trusted not to decrypt individual votes.

Low-coercion Elections. In a secret-ballot election, the coercion threat must be considered: if an adversary can force a voter to cast a vote a certain way, or even to randomize her vote, the election can be bought. Remote voting is, by nature, vulnerable to coercion, since an adversary can simply watch a voter's every move or, in many practical cases, simply purchase from the voter her authentication credentials and vote entirely in her place. Some solutions to the problem have been proposed [17], but all require at least one truly private interaction between the voter and the election authority.

A number of elections can be considered "low-coercion," a term introduced by the Helios paper. In these elections, ballot secrecy is important, because people might be peer-pressured into voting one way or another, but actual coercion is an unlikely concern because voters often have an effectively private space at home, or because the stakes are not quite high enough to warrant bribes or threats. A University President election is precisely the kind of election that is likely low-coercion: most voters do not physically reside with other voters, and even if peer pressure is applied, money or bribes are usually out of scope. This provides an opportunity to achieve most of the important properties of open-audit voting without the full complexity of a National election.

Voting over the Internet. Many voting security experts believe that voting over the Internet is the most troubling of all proposed voting system evolutions [16]. Even if one sets aside the risk of coercion (as UCL did, given the low-coercion setting), there remains the problem of the end user's computer and the relatively widespread compromise of consumer operating systems, e.g. the numerous and well documented botnets [1]. Helios 2.0, like its predecessor, does little to specifically counter the threat of client-side operating-system or web-browser compromise; a specifically targeted virus could surreptitiously change a user's vote and mask all of the verifications performed via the same computer to cover its tracks.

In this deployment, UCL believed that the likelihood of such an advanced attack against the UCL election was extremely unlikely. UCL also made available a set of secured client machines for voters who wished to use an official voting machine. It should be noted, however, that UCL and the authors do not endorse the use of Helios 2.0 for large, high-stakes, governmental elections where the threat of a targeted virus would be far more realistic.

Organization. In Section 2, we detail Helios 2.0, a series of upgrades to Helios developed specifically to fulfill the needs of a large open-audit election like UCL. In Section 3, we consider the organizational aspects and specific customizations needed for the UCL 2009 Election and, in Section 4, we discuss the statistics of the election and make recommendations for improving open-audit elections at UCL and other universities in the future.

2 Helios 2.0

In response to UCL's needs, we improved Helios and released Helios 2.0, available as open-source software [1]. The details of Helios 1.0 can be found in the original paper [2], though we summarize the important points here.

Helios 1.0 is a web-based open-audit voting system. Votes are encrypted, using the *browser-based ballot encryption program*, with El-Gamal encryption of a plaintext representation of the choices. Once loaded, this ballot

[1] hosted at `http://github.com/benadida/helios`

encryption single-page web application does not access the network again until the vote is encrypted and ready for casting. Encryption is specifically achieved within the web browser using LiveConnect to access the Java Virtual Machine from JavaScript, enabling vote encryption in 3 or 4 seconds on a typical configuration. Ballot casting assurance is achieved using the Benaloh cast-or-audit voting protocol [6] implemented in part by the *ballot verifier* which ensures that an audited ballot indeed corresponds to the fingerprint generated before the cast-or-audit choice. A Sako-Kilian [22] / Benaloh [5] *mixnet* provably shuffles the votes, then the election server decrypts the shuffled votes and tallies the results. An *election verification program* downloads all encrypted votes, shuffled votes, decrypted ballots and proofs, and verifies that the election was run correctly.

Helios 2.0 uses the same techniques for web-browser-based encryption, but otherwise provides significant updates to the rest of the protocol.

2.1 Homomorphic Tallying

Helios 2.0 shifts from mixnet-based to homomorphic tallying [9], because homomorphic tallying is easier to implement efficiently, and thus easier to verify, especially when a third party writes verification code. We use Exponential El-Gamal, a variant of El-Gamal [12] where one encrypts g^m rather than m in order to achieve an additive homomorphism, because it is easier to implement than alternative additively homomorphic schemes such as Paillier [20]. (Decryption requires a discrete logarithm computation, though with a relatively small exponent that ensures that the computation is, in fact, quite tractable.) In addition, El-Gamal lends itself quite easily to distributed decryption with joint key generation, where other additively homomorphic systems like Paillier are significantly more complicated. For simplicity and ease of discrete-logarithm computation, we use a single ciphertext for each option of each election question, rather than attempt multi-answer encoding with the more involved proofs of correct ballot form [4]. A ballot is then composed of:

- a ciphertext for each available answer to each election question,
- a disjunctive zero-knowledge proof [10] that each such ciphertext encodes either a 0 or a 1, and
- a disjunctive zero-knowledge proof that the homomorphic sum of all ciphertexts for a given question is the encryption of one out of $0, 1, \ldots, max$ for a pre-set maximum (ensuring that between 0 and max answers are selected for each question.)

In order to achieve high security with efficient modular exponentiation, all operations were performed in a subgroup of order q of $\mathbb{Z}_p^*$ where p and q are 2048 and 256 bit long primes, respectively.

Tallying and Election Verification in the Browser. In order to make the simplest use case as easy as possible, Helios 2.0 includes a web-based *election tallying program* and *election verification program*, using the same Javascript & HTML technology as the ballot verification program. This approach does not scale well to more than a few dozen votes, given browser limitations, so UCL implemented its own, high-speed, offline tallier and verifier (See Section 3).

2.2 Distributed Decryption

Another significant update to Helios was the addition of distributed decryption to ensure that multiple trustees are required for decryption. This ensures that only the homomorphic tally of all votes is decrypted, never an individual ballot. Two options were available:

- having each trustee generate a typical El-Gamal public key using the same (p, q, g) parameters, and combining the public keys using simple multiplication, or
- having each trustee generate a typical El-Gamal public key using the same (p, q, g) parameters, then have each trustee generate and publish a Lagrange coefficient to enable threshold decryption [13, 21].

Because the second approach is a little bit tricky to implement securely, with an additional step in the interaction between trustees to generate the Lagrange coeficients, we opted for the slightly less robust, but just as secure, first option. UCL used a notary public to handle backups of keys to ensure robustness (see Section 3.) We note that robust key generation, though crucial, is not entirely novel in this space: at least one other voting system, ADDER [18], already implements distributed key generation.

2.3 Machine Interface for Modular Authentication

By default, Helios authenticates users by email address and an election-specific password. This password is generated by Helios upon voter registration and sent by email to each participant. For UCL and other large institutions which already have an institution-wide authentication infrastructure, a better approach would be to plug into the pre-existing authentication infrastructure.

To enable this approach in a modular fashion, Helios 2.0 enables a separate, trusted server to submit ballots on behalf of voters. Thus, this trusted server can perform voter authentication however it sees fit, then submit the results of this authentication action to Helios 2.0 at ballot submission time. We use the standard oAuth protocol [19] (in so-called "2-legged mode") to authenticate a call from the UCL authentication system to the Helios backend server. The use of oAuth effectively sends an HMAC [15] of the request using a shared secret between the two servers (See Figure 1.)

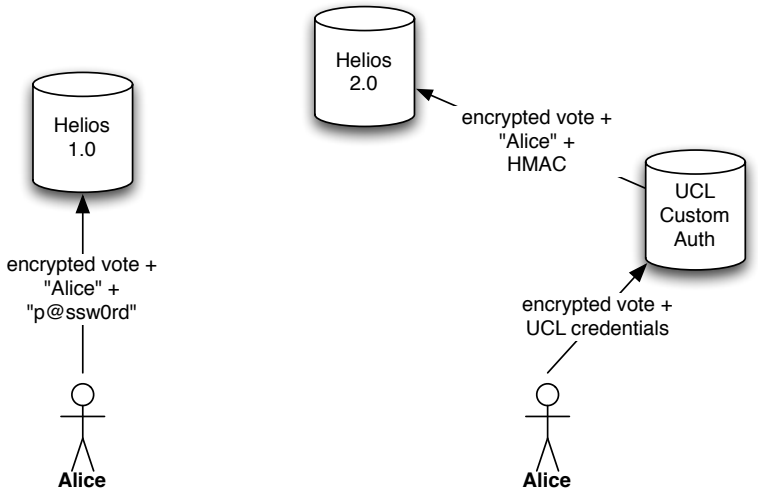

Figure 1: An API for modular authentication in Helios 2.0.

It occurred to the team that a better long-term design will likely be the separation of the ballot preparation server from the ballot submission server, so that preparing the ballot is independent of the eventual authentication required at submission time. We did not have time to implement this modular architecture for the UCL election, but we intend to do so with Helios 3.0.

Security Implications. If a server is able to authoritatively claim that a ballot comes from a particular voter, then there is always the chance of ballot stuffing. In fact, this exact same problem exists with Helios 1.0: the Helios server itself could stuff the ballot box near the end of the day. In the case of an open-audit voting system, as described in the original Helios paper, the simple defense against this is to ensure that the voter list is published at the end of the election for verification during an audit phase. It is expected that ballot stuffing would be detected at this time.

In the case of Helios 2.0, the additional server is not introduced for security reasons, but rather for modularity of the authentication implementation. The trust relationship hasn't changed: we expected the set of Helios servers not to stuff the ballot, and we verify this during the audit phase and after the election results are published by publishing the voter list.

2.4 Standalone Open-Source Distribution

Helios 1.0 was built against the Google App Engine [2] "cloud"-based hosting environment, with the goal of rapid development, deployment and scalability. Though, in theory, an open-audit voting system does not depend on the individual or organization hosting the software, there are a number of reasons to make the Helios software independent of a proprietary platform:

[2] `http://code.google.com/appengine/`

- some organizations have policies against running software outside of their network,
- some organizations, in particular UCL, are not US-based and are subject to laws that complicate the use of a US-based hosting service,
- though it isn't technically required in an open-audit system, many people are more comfortable with a completely open-source codebase they can install themselves.

For these reasons, Helios 2.0 was built to run on a completely free/open-source software stack, using the Python programming language [26], the Django web toolkit for Python [24], and the PostgreSQL database [25].

2.5 Improved User Interface

The Helios 1.0 user interface was functional but usable mostly by people with a deep understanding of cryptographic voting. In Helios 2.0, the interface was modified to take into account that many users would not be voting experts:

- A progress bar along the top of the voting interface keeps the user informed of how many steps they've completed and how many are left to complete.
- Technical language, in particular as it pertains to the audit phase, was cut out in preference of more typical voting language.
- During cryptographic operations that can take a few seconds, a "please wait" message was added to prevent user confusion over whether the application "hung."
- Significant testing across browser platforms was performed to ensure proper Java capability for cryptographic operations and notify the user in case of a compatibility issue.

The test election at UCL, detailed in Section 3, helped debug and improve this user interface with the help of a few thousand users, the UCL help desk, and a number of in-person user-observation sessions.

2.6 Additional Tweaks

A number of small tweaks and feature updates were added to Helios 2.0 to fulfill UCL's needs.

Minimum Number of Answers. Helios 1.0 allowed for multiple answers to a given question, up to a pre-set maximum. The UCL election was set up to require an active choice, with "blank vote" one of the options. This ensured that voters would not cast a blank vote by mistake, given that blank votes can cause a stalemate by contributing to the number of votes needed to gain an absolute majority, without contributing to any candidate's total. This feature was implemented by allowing an election designer to designate a *minimum number of answers* for a given question. Then, the proof of ballot correctness was tweaked to limit the number of clauses in the disjunctive zero-knowledge proof so that the sum of all ciphertexts must be one of $min, min + 1, \ldots, max$. In particular, if min is set to 1, then at least one option must be selected from the available answers.

Standalone Verifier. The Helios 1.0 ballot verifier, written in HTML and JavaScript, depended on the Helios backend server to look up the election parameters and ensure that a ballot was properly encrypted. In Helios 2.0, we built a standalone, static HTML & JavaScript ballot verifier that can be deployed on any web server with a given election's parameters wired directly into the verifier file. This allows a number of parties to publish a web-accessible verifier, so that voters have their choice of trusted entity to verify the proper encryption of their ballot. In the UCL election, Harvard and the École Normale Supérieure de Cachan hosted such verification programs (see Figure 2.)

Formal Verification Specifications. In order to facilitate the creation of third-party verification tools, Helios 2.0 provides a formal specification document of the verification procedure, including the documentation of all data models and algorithms. The specification can be found at `http://www.heliosvoting.org/`.

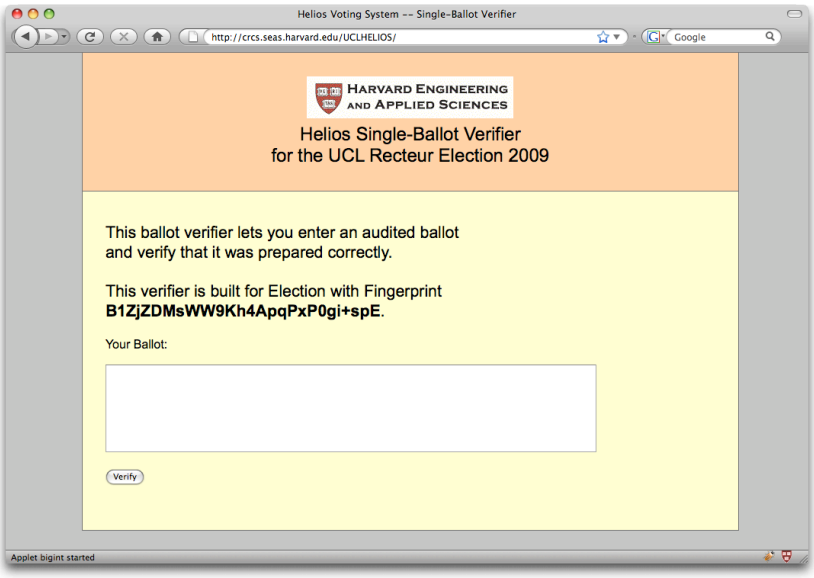

Figure 2: Harvard hosts a ballot verifier for the first round of the UCL Election.

Substantiation of Complaints Helios 1.0 provided an unauthenticated web bulletin board, which might open the path for unsubtantiated complaints by voters who either did not correctly record their vote receipt, or would like to raise suspicions about the validity of the bulletin board content. Therefore, we decided to provide voters with a document digitally signed by the voting server for each important stage of the election: at registration time, and before the web bulletin board audit phase. These documents would provide each voter a way to prove to the election commission, if it is ever needed, that she actually register for the election, but that this registration was lost by the registration server, or that a vote that was verified during the web bulletin board phase was modified in the tallying phase. While these signed documents allow each voter to substantiate potential complaints, they also allow the election commission to require the voter to provide these documents and to decide accordingly the credit that needs to be given to such complaints.

Providing the voters with signed documents raised the question of the verification of these signatures. The verification of signed pdf documents appeared to be the most usable procedure in the UCL IT environment.

3 Organization of the UCL Election

3.1 Election parameters generation

Key generation. In an election based on homomorphic tallying, the confidentiality of the votes fundamentally relies on the secrecy of the private key that is used to decrypt the tally, but that could also maliciously be used to decrypt individual votes. In order to improve the control of the decryption process, the election private key was generated and stored in a distributed way, by trustees selected from various voter groups (students, administrative staff, . . .), but also from the invited external experts from the election commission.

Producing keys in a distributed way for trustees raises interesting challenges, however, specifically when the trustees do not have a background in computer science: how can we convince the trustees that the key they are producing is really random and not chosen in advance, that no extra copy of the key is made, or that the outputs they provide at tally time do not also contain their private key (hidden in the randomness of some proofs, for instance)?

Our solution to this conundrum was to distribute the trust between multiple parties without requiring trustees to become experts themseles. A meeting with the election commission was organized, together with several experts in computer science. Several laptops were brought, from which the hard disk drives were removed and wireless network cards disabled. Then, those laptops were boot up using standard linux live-CDs, and the key generation code, kept as simple as possible, was loaded on the machines through USB sticks, after inspection by the external experts. Keys were then generated and stored on the USB sticks, the laptops shut down, and the linux live-CDs destroyed. A similar procedure was followed when those keys were used for decryption.

Though this ensured that no single party could easily steal the trustees' decryption keys or decrypt individual votes, it still required the trustees to place significant trust in the election commission to protect individual voter privacy.

Publication of election parameters. Helios elections are uniquely identified by a cryptographic hash of the different parameters that characterize the election, including the public key that is used for encrypting votes, the election questions and possible answers, and the minimum and maximum number of answers that can be provided for each question. Once the key generation phase was complete, the election identifiers were computed and published (see Figure 3) in a digitally signed pdf document. Election identifiers are then re-computed and displayed in the voter browsers at voting time, offering the voter a verification opportunity.

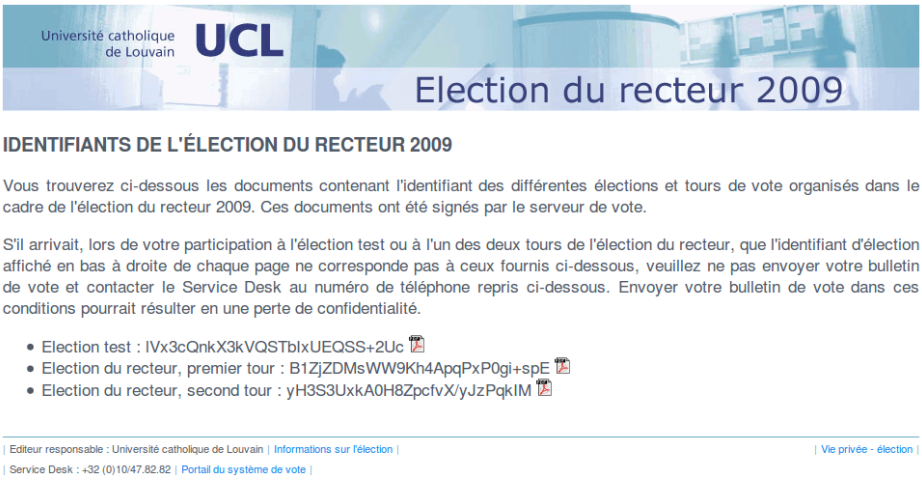

Figure 3: Election identifiers are published and available in signed pdf documents.

3.2 Registration and Voter Authentication

Motivation for the registration phase. Voting for the University President of UCL is not mandatory. The major problem with such elections is the potential risk of ballot stuffing, especially when the voter turnover is not expected to be overwhelming. In addition, when using a remote web-based system, authentication of voters is particularly critical. To mitigate this risk, a registration phase was set up. This ensures that only the eligible voters who are active in the process got a credential to vote, while providing time to audit the registration phase before the voting phase begins.

Use of voter aliases. The registration phase was also used to provide anonymous aliases to the registering voters, which simply were made of the letters "ER" followed by six digits. The purpose of those aliases was twofold:

1. The UCL election bylaws require the mere fact of voting to be confidential. Even though keeping the voter names secret can be seen as an obstacle to election verification (voters have no way to check whether all the votes printed on the bulletin board correspond to real voters, they can only check their own vote and the global verification is delegated to the election commission), the use of aliases simplifies some privacy issues too: the voting server, which was hosted outside UCL, did not contain any meaningful list of election participants.

2. Printing aliases on the bulletin board instead of voter names provides a second layer of confidentiality: someone who would eventually manage to decrypt the votes that are published during the election audit (by cryptanalysis for instance) would still not be able to know who people voted for, as this decryption would only reveal who a person with a given alias voted for, the link between the voter and her alias remaining confidential.

Registration procedure. The registration was organized as follows:

1. Every potential voter receives an email with an invitation to register. This email contains a link to the secure registration website.

2. On the website, the registrant authenticates herself using her university authentication credentials.
3. The registration website assigns a unique voter id (random-looking alias) to the registrant. A random password for casting votes on the voting website is also generated. Those credentials are immediately available for download in a pdf file, signed using a key certified by the UCL central authority (see Figures 4 and 5).
4. A notification of this registration is sent by email to the registrant.

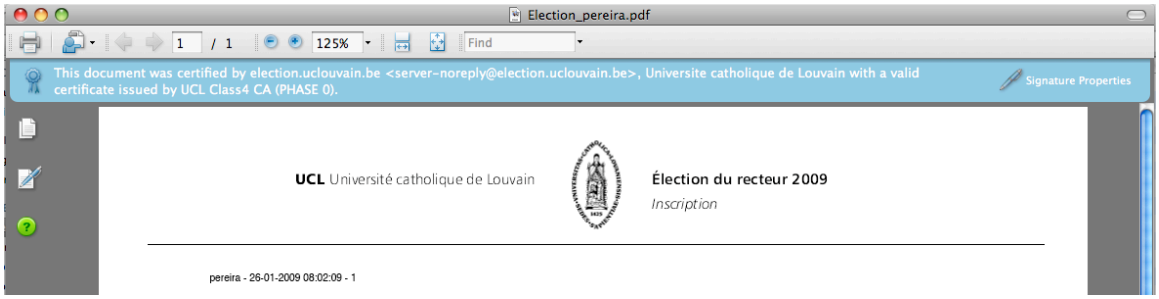

Figure 4: The signed registration PDF

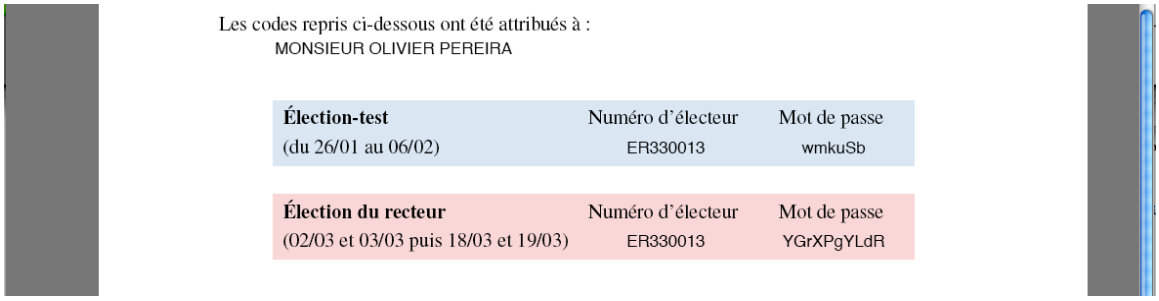

Figure 5: Passwords delivered in the signed registration PDF

The purpose of the registration notification email is to invite a voter who did not register but had her credentials used by someone else to immediately contact the UCL Service Desk in order to take the appropriate reaction. At any time after registration, voters are able to regenerate their password, following the same procedure (however the voter id is attributed once and for all.) Sensitive data such as voter id and password are never sent by email.

3.3 Voting phase

Voting through the Internet is not a common process in Belgium, and the voters included some proportion of persons who are not familiar with the use of a computer. Several dispositions were taken in order to avoid the familiarity with Internet to become a discriminant factor:

- The election was widely announced on the campus, including through non electronic medias (university newspapers, . . .).
- Public demonstrations of the system were organized in auditoriums during lunch time.
- A test election was organized in parallel with the voter registration phase, so that everyone could test the voting system before the "real" election days.
- Registration and voting offices were made available in different places in the university, in which voters could receive help from election officers.
- Printed documentation was made available, together with videos showing how to register and vote.

- Re-voting was authorized: voters were allowed to vote as many times they wanted, with the last vote being the only one counted and displayed on the web bulletin board. This feature allowed uncomfortable voters to re-vote in case of doubts, or to ask the help of anyone for submitting a random ballot, and then re-voting privately afterwards.

Figure 6: Overview of the voting interface: example of candidate selection.

3.4 Web bulletin board audit

Once the voting phase was over, the web bulletin board containing all the voter ids and vote hashes was frozen: a signed receipt was published for each vote, together with a signed version of the full bulletin board content. Individual receipts were provided to ease the use by voters (see Figure 7), while the full bulletin board was also provided in one file in order to make sure that the election organizers could not take a chance to modify votes whose receipts were never accessed after the audit day (by consulting the web server logs).

After publication of the signed receipts, a full day was devoted to the bulletin board audit. Voters were invited to consult the bulletin board and to produce a new ballot and introduce a complain to the election commission in case of disagreement with the signed data. At least two reasons could cause this disagreement: (i) the voting server was hacked (or the election organizers were cheating), and the signed vote was indeed incorrect, or (ii) the voter did not take proper care of her credentials, and a third party stole them and used them to submit a vote.

Those two reasons motivated the use of a separate authentication mechanism for submitting complaints: while ballots were still produced using the voting interface used during the voting phase, the ballot submission procedure was modified. After identification and transmission of the vote, the vote was placed in a quarantine area of the server (not visible on the bulletin board), and a signed receipt was made available for the voter, containing the vote hash. The voter was then invited to print the receipt, hand sign it, and fax it to the election commission. The motivation for this procedure was as follows:

- The production of a digitally signed receipt by the voting system gives a way for the voter to prove that she indeed transmitted a ballot to the voting system, with a given vote hash.
- The hand signature of the printed receipt provides the election commission with strong evidence of the identity of the complaining voter. In case of doubts, authenticity could be verified easily by contacting the voter and a false hand signature would be universally considered an important misconduct.
- The use of a transmission by fax (or personal deposit to the election commission) effectively produces a receipt, preventing the election commission from denying the existence of the complaint.

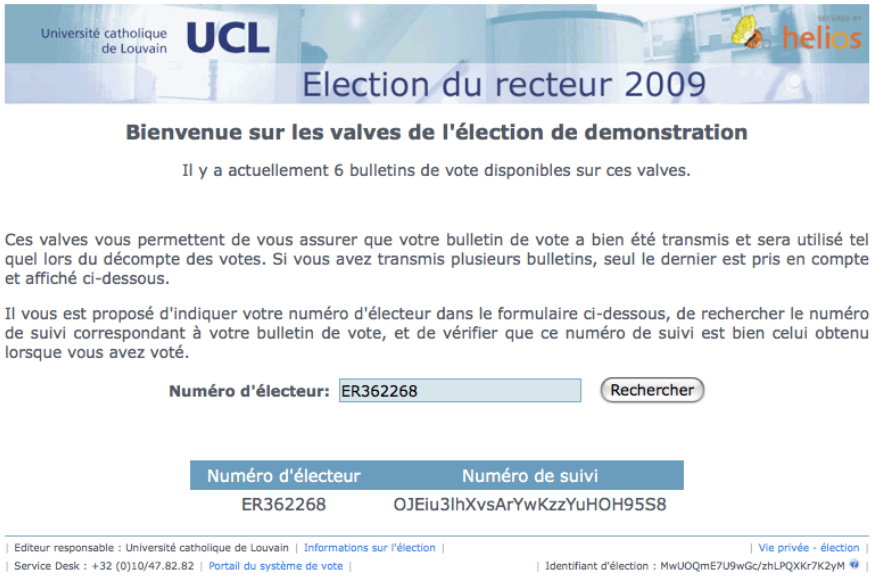

Figure 7: Overview of the web bulletin board.

- The complaint procedure was deliberately made more heavyweight than the normal voting procedure, in order to avoid having voters consider the audit phase a mere extension of the voting period.

At the end of this bulletin board audit phase, the election commission examined the complaints. They were all easily resolved and did not raise any suspicion about the behavior of the voting system, and a final, definitive version of the web bulletin board was made public and digitally signed.

3.5 Tally

Electoral system. A voter in the UCL President Election belongs to a category: faculty, staff, researcher or student. Because of the huge difference in the number of voters of those categories, i.e. twenty times fewer faculty members than students the sub-tally from each category is weighted before the final tally. According to the by-laws of the election, the weighting is a function of the turnout of each category, computed according to a fairly sophisticated formula. A candidate needs to obtain the strict majority of the weighted votes, including the blank votes, to be elected.

A knapsack problem related to privacy of the votes per category. Election by-laws require that no partial tally ever be revealed. While a standard paper-based election scheme would not completely preserve the secrecy of the election result per category (one would have to compute the tally per category, then apply the weights, and derive the final result from the weighted partial tallies), the use of homomorphic encryption allows computing the final weighted tally directly from the ciphertexts: one can multiply any plaintext message by any integer, by computing the corresponding power of the ciphertext. However, those integer computations lead to an increase in the message size, since the integral weights to be applied could be as high as 2^{45} if most voters took part to the election. (The surprisingly high value of this weight is due to the complexity of the UCL weighting formula, and the fact that only integers can be computed under the covers of encryption, which requires multiplication by the denominator of the weight fraction.)

Unfortunately, directly proceeding to the weighting of the ciphertexts may not hide the partial results: the outcome itself could reveal the per-category breakdown. Let's define w, the vector of weights (w_j is the weight of category j), and t, the vector of weighted tallies (t_i is the number of votes, after the category weight has been applied, received by candidate i). As those data are public, retrieving how a category voted involves solving the following knapsack-style problem:

> With I the number of candidates, J the number of categories and V_j the number of voters in category j:
> find a matrix $A = [a_{ij}],\ 0 \leq i < I, 0 \leq j < J$ such that $A \cdot w = t$

under the constraints: $\forall j, \sum_{i=0}^{I} a_{ij} = V_j$.

Unfortunately, with 4 categories of voters and weights that were typically bigger than the number of voters by an order 2^{30}, this knapsack problem has, with very high probability, a unique solution which can be computed easily. Therefore, even though the tally was only computed after homomorphic weighting and aggregation of the votes, it is fairly easy to derive the election result per voter category simply from the final tally.

In order to solve this problem, approximate weights were applied. Once the official weights could be computed (that is, when the web bulletin board content was completely frozen), approximate values for those weights were computed. The selection of those approximate weights was a compromise between two constraints:

- the weights needed to be high enough to guarantee good-enough accuracy on the election tally, and
- the weights needed to be small enough to guarantee that the knapsack-style problem corresponding to the per-category vote tallies had a sufficiently large solution space to prevent the computation of any useful information from the final tally.

By selecting the weights carefully, we managed to keep the error on the weighted tallies lower than 10^{-4} (with more than a thousand weighted votes distributed between the candidates) while introducing around 10^5 solutions to the per-category tally knapsack-style problem for each candidate. The solutions are well distributed on the entire solution space, so that the actual distribution of votes per categories cannot be effectively derived from the set of possible solutions.

Our solution is however tightly related to the specific election parameters we had at UCL. A more general theory would be quite useful for other similar elections.

Efficient Implementation of the Tallying Phase. The tallying procedure involves several computationally intensive tasks. The most expensive parts of those tasks are:

(i) the verification of the zero-knowledge proofs included in each vote, and
(ii) the discrete logarithm extraction providing the election result after decryption (since Exponential El Gamal encryption is used).

The tallying procedure was implemented in C, using the GMP library [14] for arithmetic operations. Our code can process 1300 ballots per minute on a standard laptop, with each ballot comprised of a vote for one question with 3 choices.) Due to the nature of the operations involved, a parallelized structure is easily deployable for further improvements, if needed.

The discrete logarithm extraction was implemented in C/GMP too, using a simple variant of Shanks' baby-step/giant-step algorithm [23]. Using a 12.5MB table, discrete logarithms of 42-bit length could be extracted in less than 10 seconds on a standard laptop. Again, this procedure could be heavily parallelized if needed, while the size of the pre-computed table could be substantially increased too.

3.6 Universal Verification

All the data needed to audit the election have been made publicly available, together with complete specifications describing the operations needed to audit the election. An independent company wrote a second instance of the audit code (in Python) on behalf of the election commission and successfully verified the election, even though the Python ballot verification procedure showed to be 100 times slower than the C/GMP implementation.

The use of two independently produced election tally and verification codes, written in two different programming languages, relying on different arithmetic libraries, provided strong evidence of the correctness of the decryption process.

4 Election-Day Statistics and Lessons

4.1 Election statistics

The Test Election. To validate the voting system deployment, we organized a test election during the registration period. After registration, any voter could familiarize herself with the system and vote for one out of five student projects that the university accepted to sponsor (or cast a blank vote.) On a voluntary basis, nearly 3000 voters took

Table 1: Results of the 2 rounds of the election (weighted votes)

	Pr. Vincent Blondel	Pr. Bruno Delvaux	blank votes
1st round	477.33	555.44	78.23
2nd round	465.74	590.48	40.78

part to this election. This test was especially useful to detect incompatibilities between the voting system and specific computer configurations. The voting website was successfully tested on Linux, Mac OSX and Windows with the Firefox 2 and 3, Internet Explorer 6, 7 and 8, Safari 2 and 3, and Chrome 1 web browsers. The test election also provided interesting information about the load to be expected on the election servers during the real election rounds (which lasted 35 hours each), and the questions received by the UCL Service Desk suggested improvements to the voting interface.

The Real Election. For the actual UCL President election, voters were asked to choose between two candidates or cast a blank vote: the ballot contained one question with three choices. Each round of the election was organized on two consecutive days (from 8AM to 7PM the next day, without interruption.) In the first round, the leading candidate missed the absolute majority required to be elected – by less than 2 weighted votes. The audit of the tally by an independent company led to a quick acceptance of the election outcome, avoiding any recount. Two weeks later, a clear majority arose out of the second round (see Table 1.)

Among the 5000 registered voters, around 4000 voters participated in each round, with a peak voting rate of 17 votes/minute shortly after a reminder was sent to the registered voters on the second election day morning. No end-of-election rush was observed. Around 1% of the voters used the re-voting feature.

In order to insure the availability of the system for every voter, polling stations with tested computer configurations were set up in different places on the university campus. Those polling stations were opened on both days of each round of the election, allowing the voter to choose between going to a voting office as is typical when participating in an election, or cast a vote using any computer at hand. It turned out that more than 97% of the voters chose the latter solution, and very few votes were cast from a polling station. It seems, from anecdotal evidence, that those who made use of the polling stations were principally not confident with the mere use of a computer.

Web Bulletin Board Audit. Approximately 30% of the voters checked their vote on the bulletin board during the election audit day, and 7 voters introduced complaints during the two election rounds. After investigation, those complaints resulted from: (i) voters who acknowledged that they did not submit a vote during the election day and took the opportunity of the audit day to introduce a vote, (ii) voters who wanted to test the system, by curiosity, and (iii) voters who inadvertently switched their vote receipts. No complaint of nature to introduce doubts about the behavior of the voting system were introduced.

Support. The UCL Service Desk raised around 120 tickets regarding the election during the two election rounds. Voters contacted the Service Desk because (by order of frequency): (i) they lost their credentials (UCL or election related) and did not know how to reset them, (ii) they used a computer without a Java plugin installed in their browser, or with a very old browser, or (iii) they were not sure that they understood the election rules or how to use the voting system.

4.2 Reaction to the use of a new type of voting system

The deployment of an open-audit voting systems brings a number of small changes in the voting process, which inevitably makes voters suspicious. Numerous presentations were given and discussions fostered make voters more familiar and comfortable with the system. The features that were the topic of significant questions and debates were: (i) the possibility to revote, (ii) the identification of the voter at the time of ballot submission (instead of before the ballot creation), and (iii) the point of the bulletin board audit day (which delayed the announcement of the election results). From anecdotal evidence and the small number of complaints, it appears eventual acceptance of the system was quite high.

Given our experience, we believe that, when one decides to introduce a new voting system, mimicking the current system in place is typically not appropriate. Instead, one should introduce all changes as early as possible in the

migration, thereby avoiding the shortcomings of repeated changes and giving voters the maximal amount of time to become accustomed to all changes.

4.3 Open-audit and the right to complain

Open-audit election empowers the voter to complain. Often seen as a risk of potential denial of service attacks, it turned out not to be the case. We believe this is due to the two following reasons:

(i) Complaints can be traced: server logs could be extracted and analyzed;

(ii) By giving each voter access to signed receipts for any interaction she has with the voting system, we give her the possibility – and therefore the obligation – to present proofs and arguments for her complain. While allowing for legitimate complaints, this largely mitigates the risk of fake complaints.

Of the few complaints we did receive, the baseless complaints were easily countered by viewing the logs and noticing that the voter's claims were, in fact, excuses for having missed either the registration or voting period.

4.4 Trustees and Cryptographic Expertise

We found one particularly difficult issue: the key generation and partial decryption by the trustees. In an ideal world, each trustee would consult their own cryptography expert and develop their own source code for key generation and partial decryption. In fact, it is likely more important for trustees to develop their own key generation code than their own verification code, since verification can be performed many times, while safe key generation and partial decryption must be done right the first time around. Unfortunately, in practice, it is very difficult to expect this much expertise from multiple trustees who are selected to be as independent from each other as possible (and can therefore not be selected from the same or related CS labs).

As described earlier, our approach was to distribute the key-generation and decryption code trust among a few members of the election commission. We can imagine other models once the Helios system is further standardized: a long-standing, published open-source key generation and partial decryption package might be available for all trustees to download and install on their own computer systems. Even in this case, however, trustees need notable technical savvy. We believe finding ways to ensure a more direct line of trust for trustees is an important next step in open-audit election operations.

5 Conclusion

With a winner declared without controversy or significant number of complaints, we believe the UCL President Election of 2009, the first significant-outcome, multi-thousand-voter open-audit election that we know of, was a success. A number of lessons on open-audit voting emerged. The ease with which frivolous complaints were countered was, to us, the most surprising, in that it is a data point against the commonly held view that allowing for complaints would inevitably lead to more problems. The significant operational advantage of having no plaintext data on the servers was useful, and we expected this, though it was significantly more useful than we initially thought.

The biggest lesson, of course, is that no matter the voting system, each election is a significant project on its own. One cannot simply install a piece of software and expect an election to run smoothly. When thinking about each election as an individual project with project managers, operational constraints and costs, we find that open-audit voting is promising. Secrecy is preserved, but evidence of proper function is available at every step. Though we recommend further studies, our initial impression is that running an open-audit election is easier and more predictable than classic elections.

An important next step in applied research on open-audit voting systems will be the ability to easily customize each part of the process, for example using some kind of interactive menu for the election manager. More research and development work in that direction is required.

6 Acknowledgments

The authors wish to thank first and foremost the UCL Election Commission and Administration and, in particular, Paul Boumal, Claire Brumagne, Armand Spineux and Virginie Sterckx for their enthusiastic support and trust during

the whole process of this open-audit election. François Glineur provided interesting discussions about the weighting procedure, and independently implemented most of the election verification procedure.

The authors also wish to thank Damien Giry at BlueKrypt for numerous helpful discussions and suggestions, Steve Kremer at ENS Cachan who hosted a single-ballot verifier during the two election rounds, and Steve Weis and Paul McDonald at Google, who generously provided increased quotas with Google App Engine for our testing (even if, in the end, we decided to use a system other than Google App Engine for legal reasons.)

Olivier de Marneffe is funded by the Belgian Interuniversity Attraction Pole P6/26 BCRYPT. Olivier Pereira is a Research Associate of the Belgian Fund for Scientific Research (F.R.S.–FNRS).

References

[1] Moheeb Abu Rajab, Jay Zarfoss, Fabian Monrose, and Andreas Terzis. A multifaceted approach to understanding the botnet phenomenon. In *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 41–52, New York, NY, USA, 2006. ACM.

[2] Ben Adida. Helios: web-based open-audit voting. In *SS'08: Proceedings of the 17th conference on Security symposium*, pages 335–348, Berkeley, CA, USA, 2008. USENIX Association.

[3] Ben Adida and C. Andrew Neff. Ballot Casting Assurance. In EVT'06 [11]. Available online at `http://www.usenix.org/events/evt06/tech/`.

[4] Olivier Baudron, Pierre-Alain Fouque, David Pointcheval, Jacques Stern, and Guillaume Poupard. Practical multi-candidate election system. In *PODC*, pages 274–283. ACM, ACM, 2001.

[5] Josh Benaloh. Simple Verifiable Elections. In EVT'06 [11]. 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] David Chaum, Richard Carback, Jeremy Clark, Aleksander Essex, Stefan Popoveniuc, Ronald L. Rivest, Peter Y. A. Ryan, Emily Shen, and Alan T. Sherman. Scantegrity ii: end-to-end verifiability for optical scan election systems using invisible ink confirmation codes. In *EVT'08: Proceedings of the conference on Electronic voting technology*, pages 1–13, Berkeley, CA, USA, 2008. USENIX Association.

[8] David Chaum, Peter Y. A. Ryan, and Steve A. Schneider. A practical voter-verifiable election scheme. In Sabrina De Capitani di Vimercati, Paul F. Syverson, and Dieter Gollmann, editors, *ESORICS*, volume 3679 of *Lecture Notes in Computer Science*, pages 118–139. Springer, 2005.

[9] Josh D. Cohen and Michael J. Fischer. A robust and verifiable cryptographically secure election scheme. In *FOCS*, pages 372–382. IEEE Computer Society, 1985.

[10] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *CRYPTO '94: Proceedings of the 14th Annual International Cryptology Conference on Advances in Cryptology*, pages 174–187, London, UK, 1994. Springer-Verlag.

[11] *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/`.

[12] Taher El Gamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Trans. Inform. Theory*, 31:469–472, 1985.

[13] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. *J. Cryptology*, 20(1):51–83, 2007.

[14] GNU Multiple Precision Arithmetic Library. `http://gmplib.org`, last viewed on April 14th, 2009.

[15] R. Canetti H. Krawczyk, M. Bellare. Hmac: Keyed-hashing for message authentication, February 1997. http://tools.ietf.org/html/rfc2104.

[16] David Jefferson, Aviel D. Rubin, Barbara Simons, and David Wagner. A security analysis of the secure electronic registration and voting experiment (SERVE). http://www.servesecurityreport.org/, January 2004.

[17] 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.

[18] Aggelos Kiayias, Michael Korman, and David Walluck. An internet voting system supporting user privacy. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 165–174, Washington, DC, USA, 2006. IEEE Computer Society.

[19] Mark Atwood et al. OAuth Core 1.0. http://oauth.net/core/1.0/, last viewed on April 14th, 2009.

[20] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In J. Stern, editor, *Advances in Cryptology - EUROCRYPT '99*, number 1592 in LNCS, pages 223–238. Springer, May 1999.

[21] Torben P. Pedersen. A threshold cryptosystem without a trusted party (extended abstract). In Donald W. Davies, editor, *EUROCRYPT*, volume 547 of *Lecture Notes in Computer Science*, pages 522–526. Springer, 1991.

[22] 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.

[23] D. Shanks. Class number, a theory of factorization and genera. *Proc. Symposia in Pure Mathematics*, 20:415–440, 1971.

[24] The Django Project. http://djangoproject.com, last viewed on April 14th, 2009.

[25] The PostgreSQL Database. http://postgresql.org, last viewed on April 14th, 2009.

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