Chapter 11

Internet Voting with Helios

Olivier Pereira
Université catholique de Louvain
ICTEAM – Crypto Group
Louvain-la-Neuve, Belgium

CONTENTS

11.1 Introduction

Helios is a voting system designed to enable practical open-audit, end-to-end verifiable elections with only the support of a web browser. Since 2009, Helios has been used by several hundreds of thousands of voters from various institutions: universities (including Université catholique de Louvain [46, 121] and Princeton University [212]), associations (including the International Association for Cryptologic Research [90] and the Association for Computing Machinery, [42]) and a number of private companies.

The choice of making it possible to create, run and tally an election with only the support of web browsers has several implications. The first is to bring end-to-end verifiable elections to anyone who can access the Internet. Using Helios does not require using any dedicated hardware, installing any specific software or having a physical mail address. This is also true for the whole election audit process: all audit operations can be performed from a basic laptop. Any voter can verify that his vote is included, unaltered, in the tally, and anyone can verify that the tally is correct. These audit operations do not require any privileged access to some data or infrastructure: they only require the manipulation of public data. Furthermore, and contrary to traditional paper voting, the audit operations do not require any kind of continuous watch: there is no chain of custody that needs to be maintained in order to perform a meaningful audit. As a result, an independent audit of the tally can be performed by anyone and at any time after the end of an election, something that would be infeasible in any large-scale traditional paper election. We will elaborate on these various operations and guarantees later in this chapter.

Helios guarantees the confidentiality of the votes by using distributed encryption (since version 2.0): votes are encrypted directly on the voter's computer and, only then, are sent to the Helios server which does not ever see any decryption key, unless the election organizers decide otherwise. Decryption keys are independently generated by trustees and are never combined: no partial decryption key should ever leave the trustee's computer. As a result, corruption of *all* the trustees of an election would be needed to perform any illicit decryption.

Still, Helios inherits the important limitations that seem to be inherent to pure Internet elections – while mitigating them to some extent. First, Helios provides very limited security guarantees to voters who would rely on a compromised computer for submitting their vote. If a malware controls the computer of a voter, and if this computer is the only interface that the voter uses for the election, the computer can display anything the voter wants to see while doing something completely different in the background (e.g., submit a vote for a different candidate or relay the vote in the clear to a third party). But, contrary to non-verifiable systems, Helios offers several audit possibilities that make it possible for a voter to detect vote alterations that would be performed by a malicious computer, provided that the voter can access an honest computer at some point, a computer that does not even need to be connected to the Internet. These observations also apply to the Helios server which, if corrupted, would be forced to leave evidence of any alteration of the votes it would perform, enabling detection as long as an honest device can be used.

Second, Helios does very little to protect voters from coercion – even though some forms of coercion resistance can be obtained through external measures, e.g., by forcing voters to use Helios in the privacy of a voting booth and in the absence of cameras. But, if the election organizers let the whole interaction between the voter and the system happen in an unsupervised context (no voting booth, no in-person registration process ...), a coercer could effectively dictate his behavior to a voter, from the beginning to the end of an election, and verify the compliance of the voter to his instructions. Still, even in this case, Helios offers a limited form of protection: voters are allowed to submit as many ballots as they want, and only the last one is tallied. This feature, besides its huge advantages for dealing with voters using an unreliable Internet connection or uncomfortable with a browser interface, enables voters who would feel pushed to vote in an undesired way at a specific moment (being "trapped" in a voting party with colleagues for instance) to submit another ballot at a later time, in a safe context.

It is of course most important to keep these limitations in mind, and to decide whether they are relevant in a specific context, before deciding to make any use of Helios in an election.

11.1.1 Helios History

The initial version of Helios, designed by Adida and presented in 2008 [44], was a web-based variant of a *simple verifiable* voting scheme by Benaloh [84], which was itself inspired from a protocol by Sako and Kilian [508]. In this version, the Helios server acted as a single trustee for the confidentiality of the votes, while guaranteeing open-audit properties. The Helios server generated a pair of keys for a public key encryption scheme; the public key was used by all voters to encrypt and submit their vote from their browser; the Helios server performed a (single) verifiable shuffle of the received ballots in order to cut them from the identity of the voters, and eventually decrypted all shuffled ballots individually. The security model for Helios 1.0 was summarized as: "Trust no one for integrity, trust Helios for privacy."

In the summer of 2008, the design of Helios 2.0 by Adida, de Marneffe, Pereira and Quisquater [46] started, resulting in the Helios protocol that is still in use today, and in the first use of an end-to-end verifiable voting system in a legally binding, multi-thousand voters election, in March 2009. A first major modification of Helios 2.0 was to abandon the shuffle-based approach and to move to a simpler and much more efficient solution based on the homomorphic aggregation of votes, inspired by a protocol by Cramer, Gennaro and Schoenmakers [178]. In this approach, and by relying on an appropriate encryption scheme, all encrypted votes are aggregated (and possibly weighted) into an encryption of the election outcome, which is eventually decrypted. A second major decision was to strengthen the vote privacy model by moving to the use of a distributed encryption scheme, in such a way that no single entity or device would, at any time, be in touch with enough keying material to decrypt individual ballots. In particular, this removed the need to have any decryption key being manipulated by the Helios server at any time, therefore considerably limiting the consequences of a server compromise. The web architecture was also considerably modified, making the various components (administration, ballot preparation, bulletin board) more independent. Eventually, various features, like the possibility to publish voter aliases instead of public voter IDs on the board, were also introduced there. The details of the Helios protocol are described in Section 11.3.

Since 2009, the functionalities offered by the Helios web application were further refined, offering new authentication modes, audit features, improved interfaces, and taking advantage of the advances in the design of web browsers to maximize compatibility. These aspects are described in Section 11.2 and some are further discussed in Section 11.4. We conclude in Section 11.5 by describing several Helios variants and extensions that have been designed and sometimes actually deployed.

11.2 Election Walkthrough

We start by describing, from a functional point of view, the process of voting in, managing and auditing a Helios election.

11.2.1 Voting in a Helios Election

11.2.1.1 Invitation to Vote

The first contact of a voter with a Helios election usually happens through an email inviting to vote. This email contains a description of the election, a link to the voting booth, and a voter ID and password that need to be used in order to submit a ballot. An election fingerprint, provided as a tracking number, is also provided, and identifies the election in a unique way (we explain how it is computed in Section 11.2.2.1).

Several variants of this process are possible:

- Helios can make use of external authentication services, like those provided by Google or Facebook (of course, these external credentials are never accessible to the Helios server) or, in some cases, custom authentication services like a university SSO or LDAP service. In this case, a vote invitation will not contain credentials.
- Helios can add a voter alias in the invitation to vote. While using an explicit voter ID is a way to improve the democratic control of who submitted a ballot in an election (a notion sometimes called eligibility verifiability [353]), many organizations consider that their list of members is private, and do not want an election to be an occasion to indirectly publish a member list. In some other cases, even if the member list can be made available to the voters, the act of voting is considered to be private, due to coercion concerns. In these situations, the Helios server makes it possible to keep the voter ID private, and only publishes anonymous aliases: this is enough for a voter to track his own ballot, but not to verify who submitted the others ballots included in the tally. This approach is also sometimes adopted as a line of defense in the case of a catastrophic failure that would result in the decryption of individual ballots, or simply of the natural evolution of computing power that would make it possible to break encryption: having ballots associated with anonymous aliases and not with real identifiers may prevent a malicious party to determine who's ballot was decrypted. Other solutions to these concerns, that do not hurt eligibility verifiability, are discussed in Section 11.5.3.

11.2.1.2 Submitting a Ballot

Using the link to the voting booth, voters can submit a ballot to the election server. Helios has an open API, which means that anyone (voter, candidate, activist ...) could program a ballot preparation system (BPS) that can be used to submit ballots in any election – this has been proposed as a programming class project in some universities. However, most voters actually use the BPS provided by the Helios website, even if using a different BPS might have some advantages in terms of trust – a voter might feel more confident that his vote will be properly prepared if he uses a ballot preparation system provided by a candidate he supports.

The Helios BPS is served as a single web page: as soon as the voting booth is loaded in a (recent) browser, a voter can go offline, make his choices, have them encrypted, have all the cleartext choices and randomness erased, and only come back online in order to submit the encrypted ballot. Voting using the Helios BPS is a 3-step process.

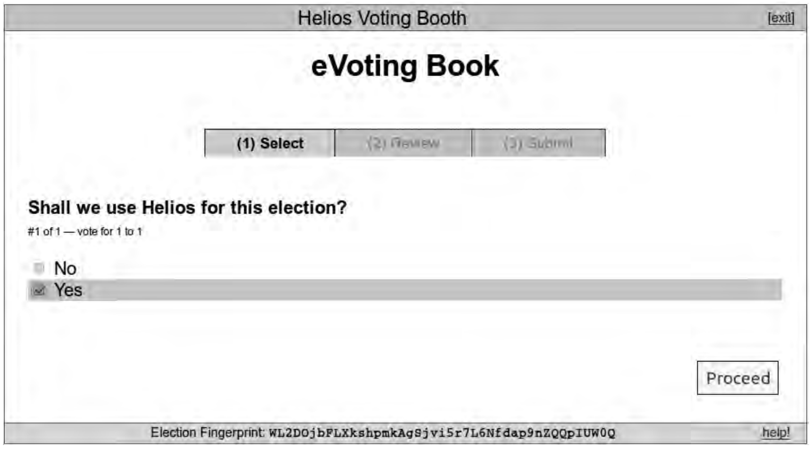
1. The voter selects his answers to the different questions that are displayed, in a form that is similar to the sample ballot from Figure 11.1. A ballot can contain any number of questions, each of these questions offering the possibility to be answered by making a number of choices determined by the election rules, e.g., one single choice, any number of choices (for approval voting), or a number within a specific range, e.g., from 0 to the number of seats available in the election. For the moment, Helios does not support write-ins or ranked voting, which would require using very different cryptographic techniques. Some Helios extensions have been used however, that can accommodate arbitrary ballot formats – see Section 11.5.1.

2. When the voter has completed his choices, he is invited to review them and to make any changes that he may desire – see Figure 11.2. In the meantime, the ballot has been encrypted, and a ballot tracker is made available to the voter. This tracker uniquely identifies the encrypted ballot while preserving the secrecy of the vote, and makes it possible for the voter to challenge the BPS and, later, to verify that his encrypted ballot has been properly recorded by the voting server and included in the tally. The voter is invited to record or print this tracker immediately.

3. The voter can then decide to submit his ballot to the Helios voting server, which will prompt him to provide his credentials and send his ballot. Requiring voter authentication at the very end of the voting process has serious advantages. It makes it possible for anyone to review the ballot style and the ballot preparation process, possibly in collaboration with other persons in case of doubts, without any fear of having their credentials stolen. It also makes it harder for a corrupted Helios server to serve a tampered version of the BPS as a function of the voter credentials (possibly targeting a specific voter population with malicious software), since the identity of the voter is unknown when the BPS is served.

It can be observed on Figures 11.1 and 11.2 that the election fingerprint, which was part of the invitation to vote, is displayed at the bottom of each screen of the BPS, for verification by the voter. This fingerprint is not simply served from the Helios server, but actually recomputed by the BPS as a function of all election parameters: election URL, ballot encryption keys, questions, answer rules ... This feature provides a safety measure for the voter and could also help detect a malicious Helios server or vote invitation when an independent BPS is used.

It can eventually be observed in Figure 11.2 that two different options are presented to the voter: either simply submit the ballot to the Helios server, or verify that


Helios Voting Booth [exit]

eVoting Book

(1) Select | (2) Review | (3) Submit

Shall we use Helios for this election?

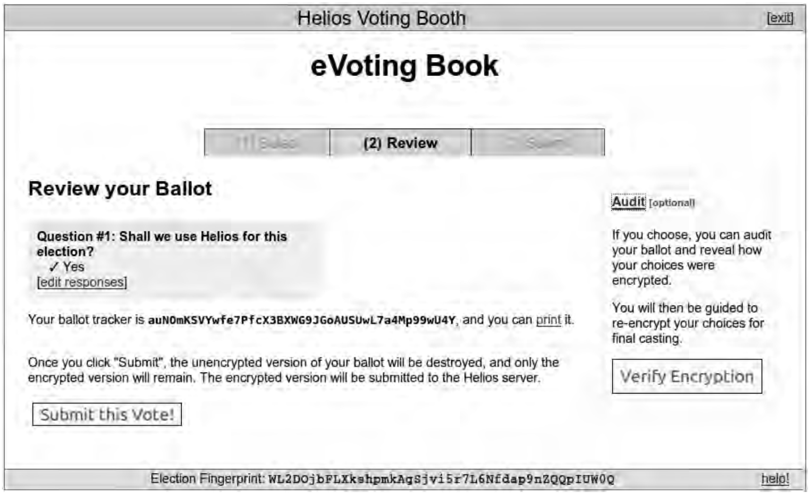
#1 of 1 — vote for 1 to 1

No [✓ Yes]

Proceed

Election Fingerprint: `WL2DOjbFLXkshpmkAgSjvi5r7L6Nfdap9nZQQpIUW0Q` help!

Figure 11.1: Selection of an answer in the Helios Ballot Preparation System.


Helios Voting Booth [exit]

eVoting Book

(1) Select | (2) Review | (3) Submit

Review your Ballot

Question #1: Shall we use Helios for this election?
✓ Yes
[edit responses]

Your ballot tracker is **`auNOmKSVYwfe7PfcX3BXWG9JGoAUSUwL7a4Mp99wU4Y`**, and you can print it.

Once you click "Submit", the unencrypted version of your ballot will be destroyed, and only the encrypted version will remain. The encrypted version will be submitted to the Helios server.

Submit this Vote!

Audit [optional]

If you choose, you can audit your ballot and reveal how your choices were encrypted.

You will then be guided to re-encrypt your choices for final casting.

Verify Encryption

Election Fingerprint: `WL2DOjbFLXkshpmkAgSjvi5r7L6Nfdap9nZQQpIUW0Q` help!

Figure 11.2: Ballot review in the Helios BPS, and choice to submit or verify its content.

the ballot has been encrypted correctly (see the frame on the right of the ballot). We will come back to this feature in Section 11.2.3.

Any voter can repeat the whole ballot preparation and submission procedure any number of times: only the last received ballot will be taken into account, and voters can check this thanks to their ballot tracker.

11.2.2 Election Management

A public Helios instance is hosted on `https://vote.heliosvoting.org/`, from which anyone can create and manage elections. This page also contains a link to the Helios code repository, currently hosted on Github, which makes it possible for anyone to host a personal Helios server. The code is released under Apache 2.0 license.

11.2.2.1 Election Creation

When creating a new election, the election administrator first defines the election name, as well as several general features: whether the election is intended to run with a closed or open list of voters, whether voter aliases are to be used (see discussion in Section 11.2.1.1), and what contact address should be offered to the voters for support. As an interesting feature, Helios also offers the possibility to have the lists of answers randomized every time they are displayed by the BPS. This feature makes it possible to remove inequalities that may result from the position of some candidates on a ballot (e.g., top of the list), when this is considered to be useful.

In a second step, the election administrator is invited to define three lists.

Questions An arbitrary list of questions can be included in a single election. For each of the questions, a list of proposed answers is defined, together with the minimum and maximum number of answers that the voter is allowed to pick. The computational complexity of ballot preparation caused the size of these lists to be a common bottleneck when Helios 2.0 was released, but this limitation largely disappeared thanks to the evolution of the browsers, which made it possible to considerably improve the efficiency of the ballot preparation process.

Voters The list of voters who are allowed to submit a ballot in an election, when it is defined in advance, can be uploaded to the Helios server as a comma-separated value (CSV) file, including the voter ID, email address and full name to be used in the email vote invitation.

Trustees The trustees are the parties that are trusted to maintain the confidentiality of the votes, and to take part in the tally decryption process. By default, the Helios server is configured to serve as a trustee, but it can be revoked, and any number of new trustees can be added. Trustees have a very sensitive role. Each of them needs to generate a pair of keys, made of a public and a secret component (further technical explanations are available in Section 11.3.2.) The public key needs to be uploaded to the Helios server, while the secret key must be kept safe and … secret. At tallying time, all trustees are required to take part in the decryption of the election results, by making use of their secret key. As a corollary of this tallying procedure, if any trustee is missing, there will be no way to obtain the election result, and it is likely that the election will need to be restarted. More robust procedures have been proposed in the literature, in order to be able to tolerate a limited number of failing trustees [457, 252], but they are considerably more difficult to use, requiring multiple rounds of interactions between the trustees, which is why Helios relies on this single-pass procedure that maximizes confidentiality – see also Section 11.5.4.

Practically, trustees are registered through the management interface, in a way that is similar to the voter registration process. They receive an email with a URL, from which they reach a web page that enables them to generate their key pair, upload the public component, save the secret one, and test whether the whole process succeeded.

When the questions and voters have been defined, and when the trustees have returned their public key, the election can be frozen. From that moment, no change can be made in any of the election parameters, and the election fingerprint is computed as a hash of all election parameters. This election fingerprint should be broadcast through various channels: depending on the elections, it has been printed in institutional newspapers, displayed on the election pages of the institution and/or included in the invitation to vote.

When the election is frozen, the voters can be invited to vote, and Helios offers a mailing mechanism that includes various templates and supports the distribution of the election credentials and optional aliases. Voters can then submit their ballots, as explained in Section 11.2.1. Every time a ballot is received, the Helios server checks its validity and keeps it for inclusion in the tally. When a voter submits more than one ballot, only the last one is used in the tally (which can be verified using the recorded-as-cast verification procedure explained in Section 11.2.3.2) while the others are archived.

11.2.2.2 Election Tally

Once the voting time is over, the Helios server computes an encryption of the election tally, by aggregating the last valid ballot received from each voter, using the homomorphic property of the encryption scheme that is used to protect the votes, as described in Section 11.3.2. This is a public operation, which anyone can perform as easily as the Helios server.

Then the trustees are invited to decrypt this tally. By connecting to the Helios trustee interface through their browsers, the trustees can download the encrypted tally, see its fingerprint, load their private key in their browser (it will never leave the browser), perform their partial decryption of the tally, and upload the result of this decryption to the Helios server. Just as for the ballot preparation, there is no need to use the Helios web interface for this purpose: another software, managed independently, could perform the same operations and submit the partial tally decryption.

No information about the tally can be obtained as long as one of the trustees did not submit its partial tally decryption. But as soon as all the trustees completed their duty, the Helios server combines the partial decryptions into the full election tally and makes that tally available. Again, this combination of the partial decryptions is a public operation.

When the tally is complete, all the information that is needed for verifying the election becomes available from the Helios server. Besides, the secret keys stored by the trustees can be destroyed: they are of no use for the audit, and this destruction decreases the risks of a key compromise in the future.

11.2.3 Election Audit

The audit of a Helios election includes three types of verifications.

1. The *cast-as-intended* verification enables any voter to obtain the assurance that the ballot he submits captures his vote intent.
2. The *recorded-as-cast* verification enables any voter to obtain the assurance that his ballot has been properly recorded on the Helios server.
3. The *tallied-as-recorded* verification enables anyone to verify that all the valid recorded votes are included in the tally. This verification is sometimes separated into the notions of universal verifiability and eligibility verifiability [353].

Together, these verification steps provide what is often called *end-to-end verifiability*.

11.2.3.1 Cast-as-Intended Verification

The ballot tracker displayed to the voter before the submission of the ballot (see Figure 11.2) is expected to be a faithful fingerprint of the encrypted voter intent. A voter can however legitimately question this, and be willing to find out whether this ballot really captures his intent.

Helios offers the possibility to challenge the ballot preparation system through a process that is often referred to as a *Benaloh challenge*. The key ingredient of this challenge lies in the moment at which the ballot tracker is displayed to the voter, that is, after the completion of all ballot preparation tasks (the full ballot information is in there), but before the voter authenticates to submit his ballot. This means that, when the BPS displays the ballot tracker, it does not know whether this ballot is intended to be posted to the Helios server, or if this ballot preparation is just part of an audit. The Benaloh challenge proceeds by offering the voter two options: either to authenticate and submit the ballot as described in Section 11.2.1.2, or to require a ballot audit and verify the encryption by clicking on the button in the frame in Figure 11.2.

If the second option is chosen, the BPS is required to provide all the data it has used to prepare the ballot, including all the randomness that has been used for encryption. Based on these data, the voter can now use a single ballot verification software to verify that the ballot matches its vote intent, is valid, and matches the committed ballot tracker. Since the BPS had to display the ballot tracker before the voter marked its intent to audit the ballot, it cannot adapt its behavior when the audit is requested: the data that is provided must match that ballot tracker.

Helios provides a single ballot verifier, accessible as a single web page but, as usual, there is no need to use software included in Helios for that task. Another possibility that the Helios web interface offers is to post the audited ballot on a public ballot tracking center. There, the ballot will be verified by the Helios server and by anyone volunteering to do so. This may simplify the task of the voter who then does not need to run any verification software by himself, but only to check on the ballot tracking center whether his ballot has been declared valid for the correct vote intent and ballot tracker.

In any case, a ballot that is audited cannot be submitted anymore, since the access to the audit data could violate the privacy of the vote. The goal is to verify whether the BPS behaves honestly on a specific device, and not to verify whether the specific ballot that a voter wants to submit has been correctly prepared.

We may then wonder how effective can this inherently probabilistic procedure be. The high-level response is as follows: suppose that a malicious party manages to corrupt the BPS and wants to influence the election outcome by flipping 1% of the votes (or any proportion p). Then, as soon as around 100 (or around $1/p$) ballot audits are triggered randomly during the election (by voters, activists ...), we can expect that the corrupted BPS will be detected at least once, which can in turn trigger more audits and investigations. Interestingly, the bound we mention above does not depend on the number of voters, which makes this process particularly efficient in a large-scale election. However, the effectiveness of the audit procedure really depends on the corruption model that is considered and on how the verification of the audited ballot is performed.

Corrupted ballot preparation system. The use of a corrupted ballot preparation system, originating from any source, is the typical situation in which the Benaloh challenge works: any vote manipulation by the BPS will be discovered by any honest single ballot verifier.

Corrupted Helios server. A malicious Helios server could go a bit further than corrupting the BPS, in some cases: if a voter uses the Helios BPS and the Helios single ballot verifier, then collusion can happen, and the verifier could be able to falsely convince the voter that his vote has been properly encoded. The same thing could happen if the voter decides to post his ballot on the ballot tracking center, and if nobody but the Helios server attempts to verify the ballot. This stresses the importance of using independent auditing tools when running a Helios election (see also Section 11.5.7).

Corrupted voting client. If the device that the voter uses is under control of the adversary, then the Benaloh challenge can only be effective if the voter is able to access an uncorrupted device at some point. If this is not the case, then the corrupted device will be able to display everything that the voter expects to see, whatever happens in the background. For instance, the corrupted client can prepare a ballot for candidate A, even if the voter wants to vote for B, display the ballot tracker for the A ballot and, if asked for an audit, alter the output of any ballot verifier running on the same device to make it claim that the ballot encodes a vote for B. This scenario, already described in [46], has been practically illustrated by Estehghari and Desmedt [222] for instance. In this demonstration, a malicious PDF document is forged, and is distributed as the program of a candidate. If the PDF file is opened with a specific version of Adobe Reader, a vulnerability of Adobe Reader is exploited in order to install a malicious plugin in Firefox, before making Firefox crash. Once the voter restarts Firefox, the malicious plugin becomes active and monitors the web connections of the users and performs the modifications described above to alter any vote that would be prepared in this browser in a way that would apparently pass all the verification steps as long as the voter keeps using this corrupted Firefox instance. Taking the audit data into another browser or device that would not be corrupted would be enough to detect the manipulation, but this definitely reduces the usability of the process, and is one of the reasons why we do not advise using Helios (and any other pure Internet voting system that we are aware of) in a setting where corrupted voting clients are a plausible scenario.

Corrupted TLS/PKI The access to a single ballot verifier or to the ballot tracking center will most likely depend on the availability of an authentic Internet connection. An attacker who manages to subvert TLS or a certification authority is likely to be able to provide a voter with a modified BPS and ballot verification tool, or with a modified view of the ballot tracking center, which would bypass the verification process. Similar issues can happen for the other verification procedures described below.[1]

One last difficulty may arise from this cast-as-intended verification process: the lack of evidence. If a voter claims that a ballot audit failed, there is no way for a third party to decide whether a system component is corrupted or if the voter made a mistake, either a honest one, or with the intent of raising unjustified suspicions about the system. Furthermore, even with an honest and careful voter, it might be really hard to determine whether a failed audit results from a corrupted voter device or from a malicious third party (e.g., the Helios server.) A voter could possibly try to record on camera his interactions with the voting system and his audit, but no third party would still be able to determine whether the voter device is corrupted (maybe by the voter himself) or if a third party is corrupted. So, in all cases, it will not be possible to draw conclusions from voter complaints, but such complaints will certainly be a useful alert calling for further investigations in order to collect evidence.

[1]We discuss the impact of a TLS/PKI failure here because these are elements on which Helios relies. Of course, errors in the Helios cryptographic protocols can have a similar impact. These will be discussed in the sections below.

Despite having never raised any actual difficulty in practice, this lack of an effective dispute resolution procedure creates a risk of denial of service attacks on elections, and finding practical solutions to this potential difficulty is one of the important open challenges in the area of verifiable voting technologies.

11.2.3.2 Recorded-as-Cast Verification

Once a voter is convinced that the ballot tracker displayed by the BPS correctly captures his vote intent, this ballot tracker can serve as a basis to verify the proper recording of the ballot. A voter can do that by connecting to the Helios Ballot Tracking Center web page, which displays the ballot trackers of all the votes intended to be used in the tally, and by verifying whether his ballot is actually displayed there, with the right tracking number.

Just as the access to the ballot preparation system, the access to the ballot tracking center does not require any authentication. This makes it harder for a corrupted Helios server to adapt the list of ballots depending on the voter accessing the tracking center. It also makes it easy to delegate this verification: a voter can send his tracking number to activists, or even broadcast it on social networks for verification by others. Still, the verification of the ballot presence at a given time during the election does not guarantee that the ballot will still be there, unmodified, and used as part of the tally. Therefore, it is useful that voters inspect the ballot tracking center when there is a public agreement on the ballots that will be used in the tally, that is, when the hash of the encrypted tally is made available to the trustees before decryption.

In some elections [46], an audit day has been organized before the tally: the election organizers published a digitally signed version of the ballot tracking center content, and a full day was left to the voters for verifying that their ballot was listed there. Of course, it would be still possible for malicious election organizers to sign different versions of the ballot tracking center and to target the distribution channels properly, but they would need to take the risk that someone would discover the existence of two different signed lists of ballots, which would be immediate evidence of corruption. Designs for a more robust and distributed ballot tracking center have also been proposed, including by Culnane and Schneider [185].

As for the cast-as-intended property, dispute resolution difficulties may arise if a voter complains that his ballot was not properly recorded: it may be impossible to decide whether a component of the system failed, or if the voter is trying to mislead the election organizers and participants. Solutions to this problem have been explored, either as procedures external to the normal system usage (see [46] for instance) or as internal extensions, e.g., by Culnane et al. [181] for the Prêt-à-Voter system.

Also, a voter might be subverted into connecting to a corrupted ballot tracking center, in which case verification would clearly offer no guarantee. This could happen in various ways. For instance, an attacker could modify vote invitations, making the voters believe that their ballot needs to be submitted and verified from the wrong place (note that this would also require to be able to modify the election fingerprint or to corrupt the BPS in order to avoid detection.) Alternatively, if TLS or the PKI on which a voter relies fails, a voter might have his ballot erased or replaced, and be displayed an alternate ballot tracking center despite using the correct URL.

11.2.3.3 Tallied-as-Recorded Verification

The two verification processes that we just described are largely individual: a voter checks what happens with his vote, independently of anyone else's vote.

The tallied-as-recorded verification is universal in the sense that anyone will care not only about whether his own vote was properly included in the tally, but also whether all the other votes that were included in the tally were valid votes submitted by valid voters.

The starting point for the tallied-as-recorded verification process is the ballot tracking center: from there, anyone can collect the list of people who submitted ballots and the corresponding tracking numbers. Unless the election administrators decide to obfuscate the voter names using aliases, this list provides the information that is needed to verify that the ballots were submitted by real voters. In case of doubts (voter credentials could have been stolen, or there might be ballot stuffing performed on a corrupted Helios server), it is also possible to contact voters in person, possibly at random, and to ask them to confirm their ballot tracker.

Once the list of tracking numbers is confirmed, the second step of the verification process consists in downloading the full list of ballots from the Helios server, and checking whether all these ballots match the expected tracking numbers. Based on these ballots, the validity of all the votes can be verified by inspecting the proofs that they contain. When the validity of all the ballots is verified, the encrypted votes can be aggregated into an encryption of the tally. Eventually, the correctness of the decryption of the tally can also be checked, by verifying the decryption proofs provided by the trustees. All these steps heavily rely on cryptographic techniques, which are detailed in the next section, and are detailed in the online Helios documentation, making it possible for any programmer to implement a verification system, which many programmers did.

This tallied-as-recorded verification process is considerably more demanding than the other verification steps: it certainly requires considerably more computational power, more than is typically available in a browser for any election of reasonable size. However, this verification step is also the one that can most naturally be delegated to third parties (activists, candidates), as it focuses on a global property of the election.

11.3 The Use of Cryptography in Helios

Helios makes heavy use of cryptography in order to enable the verification of an election without degrading the privacy of the votes. This section outlines the cryptographic protocols used in Helios, and discusses the various assumptions on which they rely. These cryptographic techniques are fairly close to an original proposal by Cramer, Gennaro and Schoenmakers [178]. A more detailed exposition of these techniques is available in the tutorial of Bernhard and Warinschi [100].

11.3.1 Arithmetic and Computational Assumption

The protocols implemented in Helios make use of a multiplicative cyclic group $\mathbb{G}$ of prime order q, in which the Decisional Diffie–Hellman (DDH) problem [110] is believed to be hard. This means that, given a generator g of $\mathbb{G}$ and a triple g^a, g^b, g^c where a and b are chosen at random in $\mathbb{Z}_q$, it is believed to be hard to decide whether c has also been chosen at random in $\mathbb{Z}_q$, just as a and b, or whether $c = ab$.

Among the various possible choices for $\mathbb{G}$, we opted for a subgroup of 256 bit prime order q of $\mathbb{Z}_p^*$, the multiplicative group of integers modulo a 2048 bits prime p. This choice provides us with a reasonable compromise between simplicity (no need to implement elliptic curve arithmetic), efficiency (exponentiation with a 256 bit q is approximately 8 times faster than if we choose $q = (p-1)/2$) and security (the resulting security level approximately corresponds to a medium-term protection as described in the ECRYPT report [35]).

11.3.2 Encryption

ElGamal [213] is the simplest public key encryption scheme whose security relies on the hardness of the DDH problem. It works as follows.

- The secret decryption key is a random value x chosen in $\mathbb{Z}_q$, from which the public encryption key is computed as $y = g^x$.
- A message $m \in \mathbb{G}$ is encrypted by picking a random r in $\mathbb{Z}_q$, and computing a ciphertext as $(c_1, c_2) = (g^r, my^r)$.
- A ciphertext (c_1, c_2) can then be decrypted as $m = c_2/c_1^x$, using the decryption key x.

This encryption scheme guarantees indistinguishability of ciphertexts [260] if the DDH problem is hard in $\mathbb{G}$: anyone who would be able to derive any single bit of information about the plaintext corresponding to a given ciphertext would also be able to solve the DDH problem in $\mathbb{G}$. This encryption scheme is used in Helios to protect the votes, and indistinguishability of ciphertext implies, in particular, that no one will be able to even recognize if two ciphertexts encrypt the same vote or not.

When presented a ballot with a list of candidates, a voter expresses his choices by encrypting "0" or "1" for each candidate, depending on whether he wants to support that candidate or not. This "0" or "1" are actually encoded as g^0 and g^1, which are two elements of $\mathbb{G}$, as needed for ElGamal encryption. This encoding, resulting in a scheme that is often called "exponential ElGamal," brings an extra benefit: it makes ciphertexts additively homomorphic. Indeed the product of an encryption of g^a and an encryption of g^b is an encryption of g^{a+b}. This feature is most useful for counting the votes: given a series of ciphertexts encrypting all the voters' choices regarding one candidate, we can simply multiply all those ciphertexts together, which provides an encryption of the number of voters who supported that candidate. This last ciphertext is the only one that is decrypted, which guarantees the confidentiality of the individual votes.

We however do not want to trust a single entity to not decrypt any individual vote. To this purpose, we distribute the key generation and distribution procedure among a set of trustees $T_1, \ldots, T_n$. This is performed as follows.

- For key generation, each trustee T_i generates an ElGamal key pair $(x_i, y_i = g^{x_i})$ and keeps x_i secret. The election public key is then computed as $g^x = \prod g^{x_i}$. At no point does any single party learn x.
- For the decryption of a ciphertext (c_1, c_2), each trustee T_i computes and publishes a decryption factor $d_i = c_1^{x_i}$. The plaintext is eventually computed as $c_2 / \prod d_i$.

This procedure is a simplified version of the threshold protocol proposed by Pedersen [458]. While it does not provide robustness against failing trustees, it is considerably simpler to use, as it proceeds in a single asynchronous round and does not require any private channel between pairs of trustees for key generation. In practice, some robustness can be obtained by pairing trustees in order to have at least two copies of each secret x_i.

The use of the exponential variant of ElGamal has one potential downside, though: the ElGamal decryption process actually provides the exponential encoding g^m of the message m that we want to recover, and not m. This means that an extra step is actually needed in order to complete decryption: the extraction of the discrete logarithm of g^m in base g. Given that m is typically upper bounded by the total number of voters, this extraction is hardly a problem in practice: our simple implementation of Shanks' baby-step giant-step algorithm [524] can extract a 40-bit discrete logarithm in a matter of seconds on a standard laptop [46].

If decryption efficiency had been a problem, Paillier encryption [452] would have offered a considerably more efficient solution. However, generating a Paillier key pair in a distributed way [188] is considerably more challenging, due to the need of building an RSA modulus with unknown factorization.

11.3.3 Zero-Knowledge Proofs

The verifiability of the election and confidentiality of the votes heavily rely on the use of zero-knowledge proofs, used to prove three types of statements:

1. Trustees are required to prove that they know the private key that matches the public key they are publishing;
2. Trustees are required to prove that they honestly contribute to the tally of the elections;
3. Voters are required to prove the validity of the ballot they submit.

11.3.3.1 Sigma Protocols

Helios makes use of sigma protocols to prove all these statements. We only outline these protocols here, and invite the interested reader to consult Damgård [187] for further details.

A sigma protocol defines a three-pass interaction between a prover P and a verifier V, as depicted in Figure 11.3. In the first message, called the *commitment*, P submits a list of random values a, typically made of elements of $\mathbb{G}$, to the verifier. This commitment will be used to blind the secret values about which P wants to make a statement. The second message, called the *challenge*, contains a random integer e, chosen by V. It is crucial for the soundness of the proof that P does not know e when he commits through a. Eventually, P sends the *response* f to V, which is typically made of elements of $\mathbb{Z}_q$. V eventually makes use of the proof statement, a, e and f to decide whether he accepts the proof.

$$\begin{tikzcd}
P \arrow[r, "a"] \arrow[d, Rightarrow] & V \arrow[d, Rightarrow] \\
\bullet \arrow[d, Rightarrow] & \bullet \arrow[l, "e"'] \arrow[d, Rightarrow] \\
\bullet \arrow[r, "f"] & \bullet
\end{tikzcd}$$

Figure 11.3: A three-pass sigma protocol.

We of course do not want to rely on interactive proofs in an election: the provers should be able to submit proofs of their statements once and for all and make these proofs available to anyone for future verification. This is where the strong Fiat–Shamir transformation comes into play [236, 99]. This transformation modifies the second step of the protocol, by computing the challenge e as a hash of the commitment a and of the statement to be proven, instead of having the challenge selected by V. The resulting proofs are computationally sound and non-interactive zero-knowledge in the random-oracle model [79]. Having described the common pattern of all proofs used in Helios, we turn to the actual proof descriptions.

11.3.3.2 Proving Honest Key Generation

The first context in which Helios requires proofs is during key generation by the trustees. It is clear from the key generation described above that, if all trustees collude or have their secret key stolen, then no privacy is guaranteed. We however want to make sure that, as long as one trustee behaves honestly, the privacy of the votes is protected.

Observing the key generation process described in Section 11.3.2, and assuming that we have n trustees, a malicious T_n could subvert the process to his advantage as follows. T_n would first wait until all other trustees have submitted their own public key $y_i = g^{x_i}$. Then, he would generate a key pair $(x, y = g^x)$ of his own, and publish a public key $y_n = y / \prod_{i=1}^{n-1} y_i$. As a result, the election public key would be computed as $y = \prod_{i=1}^{n} y_i$, and T_n would be able to decrypt all individual votes by himself, using the secret key x he choose.

The key element that makes this attack possible is that T_n is not required to prove to anyone that he knows x_n. And it is easy to verify that any algorithm that would be able to produce both x_n and x in such a setting would also be able to solve the discrete logarithm problem in $\mathbb{G}$ (which is harder than solving the DDH problem in that same group).

This problem is then solved by requiring all the trustees to prove that they know the decryption key matching the public key they submit. This can be done using a protocol due to Schnorr, which is depicted in Figure 11.4 (a). The commitment of this protocol is computed as g^s for a random $s \in \mathbb{Z}_q$, the challenge is made of an element of $\mathbb{Z}_q$, and the computation of the response is performed in $\mathbb{Z}_q$ as well. At the end of the protocol, V accepts the proof only if $g^f = ay^e$.

The soundness of this proof follows from two observations:

1. If, after having submitted a commitment a, the prover P is able to provide a response f that passes the proof acceptance test performed by V, then P is most likely able to do so for more than one single value of the challenge e. Indeed, if e were the only value for which P knows the correct f, then the probability of P to complete a proof successfully would be $1/q$, which would be an event that is infeasible to observe.

2. As soon as we trust that P would be able to submit two responses f_1 and f_2 to two distinct challenges e_1 and e_2, based on a single commitment a, we also trust that P knows x. Indeed, we can verify that $x = \frac{f_1 - f_2}{e_1 - e_2}$.

The intuition behind the zero-knowledge property of this protocol is the following one: V does not learn anything from this interaction (except for the knowledge of x by P) because V is able to produce an interaction that follows the exact same distribution, just by himself: V would pick e and f at random from $\mathbb{Z}_q$, then compute $a = g^f/y^e$. This zero-knowledge property is usually called *honest verifier*, because the simulation of the interaction is based on the assumption that, in a real protocol execution, V actually picks e at random, and not as a function of a for instance. This is not a problem, since we only use the non-interactive version of this protocol and model the hash function as a random oracle

$$\begin{tikzcd}
P \arrow[r, "a=g^s"] \arrow[d, Rightarrow] & V \arrow[d, Rightarrow] \\
\bullet \arrow[d, Rightarrow] & \bullet \arrow[l, "e"'] \arrow[d, Rightarrow] \\
\bullet \arrow[r, "f=s+ex"] & \bullet
\end{tikzcd}$$

(a) The Schnorr protocol proves the knowledge of the x matching a public key $y = g^x$.

$$\begin{tikzcd}
P \arrow[r, "{(a_1,a_2)=(g^s,c_1^s)}"] \arrow[d, Rightarrow] & V \arrow[d, Rightarrow] \\
\bullet \arrow[d, Rightarrow] & \bullet \arrow[l, "e"'] \arrow[d, Rightarrow] \\
\bullet \arrow[r, "f=s+ex_i"] & \bullet
\end{tikzcd}$$

(b) The Chaum–Pedersen protocol proves that d_i is the decryption factor computed with the x_i matching public key y_i and the ciphertext component c_1.

Figure 11.4: Two sigma protocols used in Helios.

11.3.3.3 Proving Correct Decryption

Honest key generation is crucial for privacy. It does not guarantee, however, that the decryption of the tally is performed correctly. A single trustee could indeed try to manipulate the outcome of an election by cheating when computing his decryption factor. Consider for instance a trustee T_i who, instead of computing his decryption factor for a ciphertext (c_1, c_2) as $d_i = c_1^{x_i}$, computes and submits a factor $d_i' = d_i/g^v$. As a result, the plaintext will be computed as the discrete logarithm of $c_2/\left(\frac{\prod d_i}{g^v}\right)$ in base g, which adds v to the correct decryption of (c_1, c_2) and v votes to a candidate. Of course, a malicious trustee will not manipulate a single ciphertext, which would increase the total number of votes and provide evidence of malfeasance (except in the case of approval voting), but will add a number of votes to one candidate, and remove that same number of votes from other candidates, keeping the total correct.

This problem can be avoided by making use of a sigma protocol due to Chaum and Pedersen [149], depicted in Figure 11.4 (b). This protocol is a kind of parallel version of the Schnorr protocol, and is used to prove that the discrete logarithm of y_i in base g is equal to the discrete logarithm of d_i in base c_1. Compared to the Schnorr protocol, the commitment is now made of two group elements, computed from a single random $s \in \mathbb{Z}_q$ selected by the prover P. Furthermore, the verifier V only accepts the proof if both $g^f = a_1 y_i^e$ and $c_1^f = a_2 d_i^e$ are satisfied. The soundness and zero-knowledge properties of this protocol follow from the same arguments used for the Schnorr protocol.

11.3.3.4 Proving Ballot Validity

A Helios ballot contains a series of questions, each with a number of possible answers. For each question, the voter is allowed to pick a number of answers, defined by the election rules: this can be just one answer, any number of answers (for approval voting) or a number within a fixed range.

For each of these answers, the ballot preparation system encrypts a "0" or a "1," depending on the choice of the voter, which makes it easy to multiply ciphertexts in order to obtain an encryption of the tally. However, once the voter choices have been encrypted, it is not possible anymore to determine what was encrypted: this is crucial for privacy. Furthermore, thanks to the homomorphic tallying technique, individual ballots are never decrypted. This very effective technique may open a new line of abuse, though: a malicious voter could try to encrypt values that are very different from 0 and 1 – say 1000 for one candidate and -999 for another. As a result, a voter might be able to add or remove an arbitrary number of votes from any candidate.

In order to avoid this potential problem, voters are required to use sigma protocols to prove the validity of their ballot. More precisely, the voters are required to prove that each of the ciphertexts they submit is either an encryption of 0 or an encryption of 1, and that the product of all these ciphertexts is an encryption of an integer lying in the prescribed range, indicating that a valid number of answers have been selected by the voter. (Of course, for approval voting, we can avoid that last proof since the number of selected answers is arbitrary.)

The sigma protocols that are used are disjunctive variants of the Chaum–Pedersen protocol, as proposed by Cramer, Damgård and Schoenmakers [176]. The Chaum–Pedersen protocol can indeed be used by a voter to prove that a ciphertext is an encryption of a fixed value v: compared to Figure 11.4 (b), y will take the place of c_1, c_2/g^v will take the place of d_i, and r, the randomness used to compute the ciphertext, will take the place of x_i.

As such, this protocol is not sufficient for our purpose: a voter needs to prove that a ciphertext encrypts either 0 or 1, without revealing which one: we actually need a *disjunctive* version of Chaum–Pedersen. To this purpose, we first observe that it is easy to generate a Chaum–Pedersen proof transcript that passes the verification procedure, even without knowing any secret, and even for a false statement: as before, we can choose arbitrary random values for e and f, then compute a_1 and a_2 in such a way that the verification equations are satisfied. This of course does not contradict the soundness of the proof: the *simulated* proof that we just produced has been computed by selecting e before a_1 and a_2, which will never happen in a real execution of the protocol. But we can exploit this strategy to produce a proof that one statement out of two is correct: the idea is to combine the computation of two proofs: one that proves that a ciphertext encrypts 0, and one that proves that same ciphertext encrypts 1. Of course, only one of the two statements can be true. Therefore, we will produce a simulated proof for the false statement, that we combine with an honest proof of the true statement.

More precisely, the prover will produce a simulated proof transcript $((a_1^{sim}, a_2^{sim}), e^{sim}, f^{sim})$ for the statement that is false, then generate commitments (a_1^{real}, a_2^{real}) to be used for proving the statement that is true. After submitting the real and simulated commitments to the verifier, the prover obtains a challenge e. From this global challenge, he derives the challenge for the real proof, computed as $e^{real} = e - e^{sim}$, and complete that proof by computing the response f^{real}. Eventually, the real and simulated challenges and responses are submitted to the verifier, who checks both proofs individually, and also checks that the sum of the challenges matches the global challenge e. Since both proofs check, and since the proof generation process is entirely symmetric from an observer's point of view, there is no way for any observer to decide which of the two proofs is the simulated one. So, by using this technique, a prover can demonstrate to anyone that he only encrypted 0s and 1s, and in a quantity that satisfies the election rules. This process can easily be generalized to prove that a ciphertexts encrypts a value that lies within an arbitrary range: simulated proofs can be produced for all the incorrect values from the range, with a real proof being produced for the correct value only.

So, while the honest key generation proof was crucial to privacy, the last two proofs that we discussed guarantee the correctness of the result: all tallied ballots are valid, and they are tallied correctly.

11.3.4 Protocol Analysis

The protocol implemented in Helios has been analyzed in a growing body of literature.

The security properties expected from Helios were outlined in the original papers: Helios is expected to offer end-to-end verifiability. The privacy of the votes relies on the honesty of at least one trustee. Moreover, the security of the voting client is also crucial for privacy: a malware recording all actions of the voters would easily violate privacy. Internally, a malicious BPS could transmit votes in the clear, in parallel with the normal ballot preparation. Coercion resistance is only offered in a very weak form, as usual for unsupervised voting systems: voters have the possibility to revote if they felt coerced to vote at some moment, but the coercer will be able to observe on the ballot tracking center that a new ballot has been submitted.

Several of these properties have been elaborated in more detail using requirement engineering techniques, including by Langer, Schmidt, Buchmann and Volkamer [359]. Volkamer and Grimm [569] also used Helios as an example in their analysis of the resilience of Internet voting systems, identifying the number of parties to be corrupted in order to make Helios fail, in different settings.

11.3.4.1 Works on Verifiability

Shortly after the release of Helios 2.0, Kremer, Ryan and Smyth [353] analyzed the verifiability of Helios, based on a symbolic model expressed in the applied pi calculus. Their analysis highlights the importance of eligibility verifiability, and stresses how the use of voter aliases changes the security model.

Küsters, Truderung and Vogt [356] also highlighted the difficulties arising from the use of voter aliases, through an attack pattern that they called *clash attacks*. For instance, they consider a malicious election administrator who would know that several voters will vote in the same way. To those voters, the administrator will distribute a unique voter alias, hoping that this will remain unnoticed. Furthermore, assuming that the person controlling the distribution of the BPS can guess when these voters intend to load their BPS, the administrator feeds these voters a modified BPS that uses fixed randomness. In this way, all these voters who share a single alias will prepare the exact same ballots, with the same ballot tracker. These ballots will pass cast-as-intended validity tests, since they are correctly built. Then, when the voters submit these ballots, they will appear on the ballot tracking center, but all under a single alias – hence the clash. Voters will not notice this, unless one of those ballots is posted online for audit, which may make duplication visible, or if they observe that a ballot already appears on the ballot tracking center for their alias before they vote. Eventually, if the attack works, the election administrator can create for himself as many fresh voter aliases as there are clashing ballots, and use them to vote freely in order to obtain the expected number of ballots displayed on the ballot tracking center. Such a scenario does not work if explicit voter ids are used, since these voter ids would prevent clashes from happening. The same paper also discusses the accountability of Helios, a strong form of verifiability that requires the possibility to identify which system component failed: it was pointed out that Helios does offer very little accountability, because most Helios operations are not authenticated by the party realizing them.

The works that we just described all assumed that the cryptographic primitives used in Helios presented the expected properties. Bernhard, Pereira and Warinschi [99], while investigating the ballot privacy in Helios, explored the lower-level cryptographic properties of the protocols used in Helios, and of the non-interactive zero-knowledge proofs in particular. As explained before, these proofs are made from sigma protocols made non-interactive thanks to the Fiat–Shamir transformation. This transformation however comes in various flavors in the literature: in the weak variant, only the proof commitment is hashed; while in a strong variant, the proof statement is hashed as well. The weak variant was used in Helios and Bernhard et al. show that, given the specific way in which these proofs are used in Helios, i.e., given that parties would be able to choose their proof statement as a function of the proof challenge, this can actually break the soundness property. Several attack scenarios are demonstrated from there. In the most important one, a coalition of a voter with all the trustees would make it possible to build a single ballot encrypting an arbitrarily chosen number of votes, in such a way that this ballot would pass all the verification procedures and even be indistinguishable from a regular ballot. In order to prevent this, the strong Fiat–Shamir transform should be adopted in Helios.

11.3.4.2 Works on Ballot Privacy

Cortier and Smyth [173], in parallel with Wikström, investigated privacy properties of Helios and observed that Helios did not do anything to prevent a voter from taking someone else's ballot (from the ballot tracking center, for instance) and resubmitting it as his own. While this is a serious privacy threat in mix-net-based elections, as demonstrated by Pfitzmann and Pfitzmann [460], the privacy impact of this possibility is much more limited in a scheme based on homomorphic tallying like Helios. Nevertheless, situations like the following one could happen: in an election with three voters, one voter could decide to copy someone else's encrypted vote, and then deduce the content of this vote from the election result. Of course, the voter who copies the ballot needs to forfeit his own vote in order to learn the vote of another party. Ballot copying can be prevented in Helios by using a non-malleable (NM-CPA) encryption scheme in order to prevent the submission or rerandomized versions of previous ballots, and by rejecting identical ciphertexts from the ballots to be included in the tally [99, 97]. In other works, ballot copying has also been identified as a useful feature of a voting system (despite its potential impact on privacy), e.g., for liquid democracy, and variants of Helios exploiting this feature have also been proposed (see Section 11.5.5).

In further works, it was shown that a large subset of the Helios protocol, using non-malleable encryption and rejection of duplicate ballot, would offer ballot privacy and independence in the sense of a simple ideal functionality [98, 96].

11.3.4.3 Miscellaneous Works

In an early work, Groth [271] analyzes the CGS protocol [178], which was a precursor of the protocol implemented in Helios. This analysis is performed in the UC framework, and shows that the CGS protocol implements an ideal voting functionality under reasonable assumptions. This analysis provides an increased confidence in the general protocol approach that is used in Helios, but the analyzed protocol also differs from the one in Helios in many sensible ways: the ballot preparation process includes the voter id, voters can submit only one ballot, the Benaloh challenge was not part of the protocol ... An interesting feature of the use of the UC framework lies in the very natural way in which the intended properties of the voting system are captured, i.e., by showing that running a protocol is as good as interacting with an ideal voting functionality that receives votes, possibly intercepted by an adversary, and computes the corresponding tally.

11.4 Web Application Perspective

Offering strong verifiability properties does not reduce the need of a high-quality software. From a security point of view, verifiability only makes it possible to detect errors, while it is of course most desirable that no error happens. The privacy of the vote also depends on the software, and is mostly orthogonal to end-to-end verifiability. Besides, the usability of a voting system is crucial, and is also often seen as a security feature.

Helios, since version 2.0, is a Django application and makes heavy use of JavaScript for most of the sensitive parts of the system. In particular, the ballot preparation system is a pure JavaScript application. The code of Helios is available on Github.[2]

A Helios server can be accessed from a browser and offers three main components: it serves the election administration interface, the voting booth, and the ballot tracking center. A single ballot verifier is also available as a separate component. Besides, a Helios server offers a public API, which can be used to access all election data (public keys, list of voters, ballots, ...) as JSON strings. This API is the main interface used by external audit tools.

11.4.1 The Browser Interface

The use of Helios confronts voters, trustees and election administrators with a number of uncommon features: the availability of a ballot tracker and tracking center, a key management process for trustees, the requirement for voters to authenticate at the end of the voting process, etc.

The Helios interface was refined on various occasions since the initial Helios design, based on user feedback and on independent studies that have been performed (see also Section 11.5.6). Being able to run several elections within a single large organization with a dedicated helpdesk was a source of particularly valuable information. In particular, it showed that many of the original features of Helios become part of voter habits very quickly.

Cultural factors also mattered in many cases: different countries use different ballot and form presentation styles in their official communications, and voters tend to express preferences for the presentation styles with which they are the most familiar. Other aspects of Helios do not have common counterparts in the life of the voters and provide a “clean slate” in terms of presentation. For instance, while it was feared that asking voters to perform extra verification steps (e.g., look for their ballot on the ballot tracking center) would be a major obstacle, we observed that voters get used to it fairly quickly, and even complain about a lack of security if they are later invited to vote with a voting system that does not offer these audit possibilities.

Still, some security features keep presenting usability obstacles. For instance, the cast-as-intended verification procedure is arguably challenging to perform. The length of the tracking numbers can also be perceived as fairly demanding when a voter needs to perform a verification. Here, the difficulty lies in the need to have digests that are reasonably efficient to compute but also guarantee collision resistance.

[2]`https://github.com/benadida/helios-server/`

While base64-encoded SHA-256 hashes are used for the moment, other representations and hash function choices might be beneficial. The secret key management process can also be fairly challenging. In order to simplify the key generation process as much as possible, Helios uses a distributed key generation mechanism that does not tolerate the failure of any trustee. These key generation and decryption processes are often performed by persons selected for their standing in the election and not for their computer expertise, which is consistent with their role but often turns out to be practically challenging, especially when the manipulation of secret data is involved. It would be a very useful step forward to design procedures that would further simplify this process and make it possible to tolerate a limited number of failures.

11.4.2 Cryptography in the Browser

Running in a browser the relatively sophisticated cryptographic operations that are needed for the preparation of a ballot proved to be a challenging task.

The early versions of Helios made use of LiveConnect to access the Java Virtual Machine (JVM) from JavaScript. The JVM provided secure randomness and offered support for expensive computational operations like modular exponentiations. This was however a major source of voter complaints, due to the unavailability of the JVM, or due to the lack or difference of support of the JVM in various browsers.

The performance of the JavaScript interpreters included in the browsers however considerably increased during the early years of Helios: between 2009 and 2011, the speed of a JavaScript modular exponentiation increased by a factor of 10 to 20, making it possible to perform the necessary computation directly inside the browser, without relying on a JVM [45]. In 2011, the SJCL library [539] was integrated into Helios, with the support of Emily Stark, Mike Hamburg, Tom Wu and Dan Boneh, which made it possible to prepare a full ballot directly in JavaScript. Workers are also used to performing most of the computation in the background, while the voters make their choices: most of the computational work that is needed to prepare a ballot is indeed independent of the voter choices.[3]

Secure randomness remained an issue for a fair amount of time. At first, entropy was collected from various sources like the movements of the mouse, and then expanded, using a mechanism inspired from the Fortuna design [235]. As an extra measure, randomness was also provided by the Helios server together with the voting booth. While this last source of entropy does not offer any protection from the server, it can provide a safeguard from the rest of the world in case of failure of the local entropy sources. More recently, the JavaScript Web Cryto API was extended to provide a source of secure randomness, which is typically collected from the system.

As of today, the Helios BPS uses JavaScript cryptography and workers for the ballot preparation, which runs quite well in the recent browsers. When an old browser

[3]For instance, when computing an ElGamal ciphertext (g^r, my^r), the two exponentiations are independent of m and represent at least 99% of the computational effort.

that does not support these features is used, the ballot preparation is delegated to the Helios server. This does not change the situation regarding the verifiability: the cast-as-intended verification just verifies a slightly different BPS. In terms of privacy, a corrupted Helios server could violate the privacy of these voters by recording their votes but, as discussed above, a corrupted Helios server could also serve a malicious BPS that would leak the voter choices anyway. This strategy then seems to provide an important usability improvement with a limited security impact.

11.4.3 Application Security

The structure of the Helios protocol considerably reduces the operational security requirements of a voting server. Regarding privacy in particular, the Helios server only needs to store public information: election descriptions, public keys, encrypted votes, and audit data. The secret keys of the trustees never reach the Helios server and, with the exception discussed above of old browsers using the Helios BPS, no cleartext vote ever reaches the server either (and no cleartext vote is ever stored there). These features makes it easy to deploy standard database replication tools for robustness, and to closely monitor the server content during an election, without fear of privacy loss. Still, active corruption or bugs on the Helios server might have very damaging effects, including data losses and the corruption of the BPS.

In a similar way, bugs in the Helios voting clients might cause corruption or loss of privacy. Several independent reviews of the Helios code have been performed and documented. Heiderich, Frosch, Niemitz and Schwenk [302] reviewed the Helios code, identified several potential attack sources (XSS, . . .), and proposed fixes that have been integrated. Pouillard [469], together with a team of researchers in Denmark, spotted that the ballot verification procedure implemented on the Helios server would accept some invalid votes: when voters are allowed to pick a number of candidates within a prescribed range, the Helios server would check the range proof, but not verify that the range for which the validity of the vote is proven matches the election definition (the proof would be valid, but for a wrong statement). This would enable a malicious voter to pick a number of candidates outside of the prescribed range, while being accepted by the Helios server – even though this fraud could be detected by any independent election verification tool.

11.5 Helios Variants and Related Systems

A number of variants of Helios have been proposed during the last few years, and some of them have also been implemented and used in elections. Several tools were also designed, that provide support for the audit of elections and for specific tasks like key generation.

11.5.1 Mix-net-Based Variants

A mix-net-based election, in its simplest form, works as follows: voters encrypt their votes submit them to a server. The votes then pass through a network of mixers who verifiably shuffle them (sequentially) in order to anonymize them (this could be seen as shaking the urn), before the distributed decryption of the anonymized votes happens.

This approach has some serious advantages. Most importantly, it can conveniently and efficiently accommodate arbitrary ballot formats, including ranked voting or write-ins, since there is no need to prove the validity of a ballot in the encrypted domain: validity can be checked after decryption. This advantage was the primary motivation for the development of mix-net-based variants of Helios.

Another potential advantage is that, from an educational point of view, the voting and tallying process of a mix-net-based election mimics more closely traditional paper elections – at least, when ignoring the technical details, which can actually be more cumbersome than in a homomorphic election process. The tallying procedure of a mix-net-based election is indeed considerably more complex: mixing ballots is a computationally demanding task, and the trustee decryption procedure now requires to decrypt at least one ciphertext per voter, instead of one ciphertext per question in the homomorphic approach. These constraints strongly support the homomorphic approach implemented in Helios when it can be used. Besides, the algorithms involved in a verifiable mix-net are quite sophisticated, making their implementation a fairly challenging task.

Two mix-net-based variants of Helios implementations have been described:

- Bulens, Giry and Pereira [121] made an efficient implementation based on the HTDH2 encryption scheme, which they designed as a variant of the TDH2 scheme by Shoup and Gennaro [528], and on a proof of shuffle by Terelius and Wikström [576, 552]. This variant has been used in dozens of elections, in universities and private institutions.
- Tsoukalas, Papadimitriou, Louridas and Tsanakas [558] later made another implementation, with a verifiable shuffle based on the Sako–Kilian [508] scheme (following Helios 1.0) which, while being considerably less efficient, is also considerably simpler. Zeus has been used in dozens of elections in Greece, including some in a highly emotional context, providing an interesting experience.

11.5.2 Variants Aiming at Countering Ballot-Stuffing

The ballots displayed on the tracking center do not contain any secure personal information: the tracking center only has public voter names (or aliases) and encrypted votes, whose preparation only requires public knowledge. As a result, a corrupted

Helios server could add, remove or modify ballots quite easily. It would of course take the risks of being detected, especially when aliases are not used: any ballot to be included in the tally needs to be associated to a voter name, which opens the risk of complaints being introduced by any of the prejudiced voters. Limiting the impact of a corrupted Helios server is still desirable.

Helios-C is a Helios variant designed by Cortier, Galindo, Glondu and Izabachène [172] that aims at reducing the possibilities of ballot stuffing by relying on a separate registration authority, which is expected to not collude with the party recording the ballots. That authority distributes voter credentials, which are then used to digitally sign ballots. This prevents ballot stuffing as long as no collusion happens (and credentials are not stolen).

An alternate solution was proposed by Srinivasan, Culnane, Heather, Schneider and Xia [537]. Here, while still relying on a separate authority, the voting process is simplified for the voters, who now only need to store a token instead of a full cryptographic key.

11.5.3 Variants Aiming at Perfectly Private Audit Data

The Helios audit data used for the tallied-as-cast verification contain encrypted votes. But any encryption scheme comes with decryption keys and it may be the case that, due to some manipulation error or some hacking, a malicious party would be able to collect enough decryption keys to recover ballots. Besides, since the security of encryption relies on computational assumptions (the hardness of the DDH problem in this case), it is definitely possible that, in the future, someone will be able to decrypt encrypted votes, either because of the availability of more powerful computers, or because of some algorithmic breakthrough that would provide more efficient methods for breaking encryption.

In order to counter these potential issues, it has been proposed to use Helios variants that would offer perfectly private audit data, or everlasting privacy towards the public. In these variants, all the data provided by the system, and the audit data in particular, are perfectly hiding in the sense of information theory. This means that, no matter what key is leaked, and no matter what computational power is available to the adversary, the privacy of the votes remains guaranteed. Still, voters need to submit information about their vote that, for a computationally unbounded adversary with intrusion or network control capabilities, may eventually make it possible to recover the votes. But this kind of attack would require a significantly higher level of preparation.

- Demirel, van de Graaf and Samarone [196] proposed a solution based on the Paillier encryption scheme and Pedersen commitments in matching groups, following a proposal by Moran and Naor [398]. Proposals based on a distributed mix-net acting on secret shares have also been made [119], in the spirit of the secret-sharing based approach of Cramer et al. [177].

- Cuvelier, Pereira and Peters proposed another solution, based on new encryption schemes called PPATs and PPATc [186]. This approach, being based on prime order groups instead of composite groups (like Paillier encryption), brings practical key generation procedures and is considerably more efficient from a computational point of view.

11.5.4 Variants Based on Full Threshold Encryption

Threshold key generation makes it possible to tolerate the failure of a limited number of trustees, which seems highly desirable for a high-stake election. Several teams implemented such a procedure:

- Cortier, Galindo, Glondu and Izabachène [170] and Pieter Maene [378] implemented variants of the Pedersen procotol for threshold key generation [457]. The key generation is integrated in the web browser and becomes considerably more complex, requiring several rounds of interaction between the trustees, but better fault tolerance is also obtained.
- Neumann, Kulyk and Volkamer [409] designed an Android Application that automates and considerably simplifies the key generation process: the trustees are required to run the App at the same time, to perform some simple verifications, and the key generation protocol executes in the background, based on peer-to-peer communication.

11.5.5 Variant Supporting Vote Delegation

The possibility to produce ballot copies, pointed to as a source of privacy issues in the work of Cortier and Smyth, has also been seen as a useful feature offered by Internet voting systems. An increasing number of elections indeed use *vote delegation*, making it possible for a voter to delegate his vote to someone else. Examples include liquid democracy, used in the German Pirate party.

Posting ballots on a bulletin board can provide an interesting way of implementing this delegation process, with the convenience that voters may not need to explicitly delegate their vote to others, but could simply submit a copy of someone else's ballot. This idea is explored by Desmedt and Chaidos [198], who propose a variant of Helios explicitly enabling ballot copies. In a first simple non-interactive variant, ballot copies can be noticed by an external observer. In another variant, the voters can produce ballot copies that are indistinguishable from any other ballot, at the cost of interacting with their delegate.

11.5.6 Alternate Helios Frontends

Karayumak, Kauer, Olembo, Volk and Volkamer [338, 339] ran a usability analysis of Helios, based on a cognitive walkthrough, from which they designed alternate Helios interfaces. These alternate interfaces were also investigated through a user study. The proposed modifications include several changes in the voting and audit process, alternate phrasing, as well as improved consistency in the voting booth design.

Further changes in the individual verification processes were proposed by Neumann, Olembo, Renaud and Volkamer [410], who discuss the possibility to delegate the recorded-as-cast verification to third parties using an Android App and a QR-code to simplify the access to the ballot tracking center.

Various other groups designed alternate Helios frontends, modifying various visual aspects (see [378] for instance), or translating it into various languages.

11.5.7 Audit Tools

Independent Helios election audit tools are definitely highly desirable: they provide more resistance to corruption, they can help detect bugs, either directly in the Helios code or in the underlying libraries, and they can even serve as a backup for election data.

- de Marneffe designed the Helios election monitor [48]. This Web2Py application, when given the URL of an election on a Helios server, polls the Helios server every few minutes to download newly submitted ballots. It provides a full bulletin board, verifies the validity of each ballot, warns about revotes, provides graphs of the voting rates, and completes the tallied-as-recorded verification when the election results are available. The Helios election monitor also provides a single ballot verifier. All the verification procedures in this monitor are based on external cryptographic libraries, improving the code independence with Helios.
- Roeder designed Aethon and Pyrios [490]. Aethon is a tool providing a web interface from which a full tallied-as-cast verification can be performed: the auditor browser picks all audit data from the Helios server, and pushes them back to the Aethon server, which performs the verifications and displays the results. Pyrios is a more recent Go library, and offers similar functionalities, except that all the verifications now run locally on the computer of the auditor.

11.6 Conclusion

Seven years after the first use of Helios in a large-scale legally binding election in 2009, Helios and its various forks are routinely used by associations and private companies to run end-to-end verifiable Internet elections, and several hundreds of thousands of votes have been verifiably tallied.

Before running this first election, we had a lot of concerns about running an end-to-end verifiable election. What would be the reaction of the voters to the verification steps that we propose? Would these verification features be dismissed or embraced? How would we be able to handle auditors raising concerns about the election, especially when knowing how little accountability Helios offers? Most of these concerns faded now. The verifiability features of Helios do not appear to prevent anyone from voting, and we found that a surprisingly large number of voters do look for their vote on the ballot tracking center. Furthermore, many voters start feeling that checking a ballot tracking center should be a natural part of any remote voting system: how could they possibly know that their vote was correctly recorded otherwise? We also faced very few auditor complaints and, in all cases, it has been possible to dismiss these complaints quite easily: the public nature of the Helios server content makes it possible to collect detailed logs without creating any risk for the privacy of the votes, and these logs showed to be very helpful for clarifying situations. Furthermore, various simple conflict resolution procedures can also be organized, taking advantage of the fact that Helios always keeps the link between voters and their encrypted vote.

Our assessment of the effectiveness of verifiability features also was nuanced. While the invitation to consult the ballot tracking center was very well received by the voters overall, we found that most organizations refuse to display a ballot tracking center containing voter names and to publish lists of voters. This makes ballot stuffing fairly hard to detect by any auditor who would not have privileged access to election data. Except for this aspect, the tallied-as-recorded aspect of the audit appears to be largely effective, with several independent tools having been built. Cast-as-intended verification remains an important challenge, however: as of today, there is no independent and stable ballot verifier available online, and voters are simply invited to use the verifier offered by the Helios server, which is an important limitation if a corrupted Helios server is part of the threat model. Besides, if a corrupted voting client is part of the threat model, the verification process requires the possibility for the voter to access a honest device at a later time, which is definitely demanding.

Similar concerns appear about the privacy of the votes: the Helios ballot preparation system remains the only convenient and largely available way of preparing a ballot, and voters therefore essentially have to trust the Helios code regarding the privacy of their vote (or read the code that they received, which is an option for very few people only). Again, this situation could be improved if some trusted organizations or candidates were offering an independent ballot preparation system.

On a positive side, we found that the requirement to provide election audit data in real time and at all steps of an election is a very effective constraint placed on election organizers: even if they suspect that some aspects of the election will not be verified, they can never be sure, and audit data need to be committed anyway. A malicious Helios server trying to stuff extra ballots knows that these ballots need to be associated to a voter, and that this is evidence of a malicious behavior if an investigation is started. In a similar way, serving a malicious ballot preparation system always leaves evidence that can possibly be collected and investigated, even more since the BPS is a human-readable script that is distributed without access control. Overall, we feel that this requirement for the organizers to commit on audit data is, in itself, already a very strong motivation for adopting an end-to-end verifiable system.

Acknowledgments

We would like to thank Vanessa Teague for her insightful comments and suggestions about this text, as well as other anonymous reviewers. This chapter also largely benefited from numerous fruitful discussions and collaborations: with Ben Adida, of course, but also with many others, including Josh Benaloh, David Bernhard, Philippe Bulens, Véronique Cortier, Damien Giry, Steve Kremer, Olivier de Marneffe, Ron Rivest, Mark Ryan, Ben Smyth, Dan Wallach, Bogdan Warinschi and Douglas Wikström.