May 2017

Practical Attacks on Cryptographically End-to-end Verifiable Internet Voting Systems

Nicholas Chang-Fong
The University of Western Ontario

Supervisor
Dr. Aleksander Essex
The University of Western Ontario

Graduate Program in Electrical and Computer Engineering

A thesis submitted in partial fulfillment of the requirements for the degree in Master of Engineering Science

Abstract

Cryptographic end-to-end verifiable voting technologies concern themselves with the provision of a more trustworthy, transparent, and robust elections. To provide voting systems with more transparency and accountability throughout the process while preserving privacy which allows voters to express their true intent.

Helios Voting is one of these systems—an online platform where anyone can easily host their own cryptographically end-to-end verifiable election, aiming to bring verifiable voting to the masses. Helios does this by providing explicit cryptographic checks that an election was counted correctly, checks that any member of the public can independently verify. All of this while still protecting one of the essential properties of open democracy, voter privacy.

In spite of these cryptographic checks and the strong mathematical assertions of correctness they provide, this thesis discusses the discovery and exploit of three vulnerabilities. The first is the insufficient validation of cryptographic elements in Helios ballots uploaded by users. This allows a disgruntled voter to cast a carefully crafted ballot which will prevent an election from being tallied. The second vulnerability is the insufficient validation of cryptographic parameters used in ElGamal by an election official. This leads to an attack where the election official can upload weak parameters allowing the official to cast arbitrary votes in a single ballot. The final attack is a cross-site scripting attack that would allow anyone to steal or re-cast ballots on behalf of victims.

We coordinated disclosure with the Helios developers and provided fixes for all the vulnerabilities outlined in the thesis. Additionally, this thesis adds to the body of work highlighting the fragility of internet voting applications and discusses the unique challenges faced by internet voting applications.

Keywords: Internet voting, end-to-end verifiable voting, cryptography, hacking, elections

Acknowlegements

First, I'd like to thank my supervisor, Dr. Aleksander Essex, for all of his support and guidance throughout my time at Western. Beginning in undergrad, Dr. Essex helped me truly realize my interest in the field large by giving me the freedom to explore all I found interesting while still helping me find my way. It is with certainty I can say I would not be here if it were not for Dr. Essex's guidance.

To my dear friend, Vanessa, for being truly exceptional in all that she does and inspiring me to do the same. An endless source of personal and academic inspiration, Vanessa reminded me of the magic in music and dance when I needed it the most. I could not have done this without her.

To my parents at home, whose endless support really make all of this possible. For taking care of things before they could even cross my mind and for always being understanding no matter the situation.

Finally, thanks to Ben Adida and the Helios team for their excellent work and for handling this situation in the most gracious manner.

Contents

List of Figures

Chapter 1

Introduction

Internet voting seems to be one of the few remaining dreams of the digital age yet to be realized and for good reason. Unlike other activities that have moved online, such as banking or retail, the security considerations that have to be made for in-person voting do not have an easy implementation in the online world. Looking at the cities and countries that have deployed internet voting such as Switzerland [62], Estonia [91], New South Wales [101], and Washington D.C. [106], there are frequently more tales of woe than success.

Even in the broader field of electronic voting, researchers are frequently skeptical about the readiness of the technology for large-scale deployment [96]. This skepticism is not driven by pessimism but by a well-informed understanding of the complex challenges involved in handling electronic or internet voting.

At a high level, these challenges relate to how any voting system has to protect voter privacy while simultaneously be open and transparent all of this on top of a robust system ensuring correctness (with indications of incorrectness) arguably above all else. Some systems attempt to provide these services and for the most part do but as this thesis demonstrates, even well-reviewed systems trusted by people in the research community, have oversights that cause a system to fail to deliver a fair, end-to-end verifiable election

1.1 Motivation

Helios voting is one of the most popular internet voting systems, claiming to have tallied over one-hundred thousand ballots, having been used by major organizations, and the subject of many research papers [86, 88, 70, 90]. When looking for an cryptographically end-to-end verifiable internet voting system to investigate, Helios seemed to be the perfect choice.

The overarching goal of this thesis is to help make internet voting more secure by (1) discovering, demonstrating, and fixing vulnerabilities; and (2) adding to the body of knowledge to help advance further development of such systems.

1.2 Contributions

This thesis makes three specific contributions to the Helios voting platform; the discovery, exploit, and repair of three vulnerabilities that would allow:

1. A dishonest election official to convincingly rig an election by arbitrarily adding or even subtracting votes while, perhaps most concerning, still providing a valid cryptographic zero knowledge proof that the election tally and thus election proceeded without error.

2. A dishonest voter could cast a "poison ballot" which would prevent an entire election from being tallied. Similar to the previous attack, the ballot had an accompanying zero-knowledge proof of correctness. Had this not been discovered and actually used in a real election, it is unclear whether the election officials would attribute the error to malice or glitch.

3. Any member of the public to create a false election containing a cross-site scripting attack that could be leveraged to steal ballots, or prevent them from being cast at all.

The discovery of the cryptographic vulnerabilities highlights a systemic failure to check the cryptographic assumptions of the underlying mathematical structure. A problem not only experienced in Helios but also in many widely implemented online protocols such as the Diffie-Hellman key exchange in Transport Layer Security (TLS) [75, 103] among others highlighting an ongoing need for better requirements in cryptographic protocols.

The discovery of the cross-site scripting (XSS) bug highlights the fragility of the web ecosystem and adds to the list of examples where small platform-based flaws can ruin an election.

These results also resulted in a publication [66] which appeared at the Annual Computer Security Applications Conference (ACSAC) in 2016.

1.3 Organization of Thesis

The rest of the thesis is organized as follows:

Chapter 2 contains an introduction to the broader field of verifiable electronic voting, what security considerations need to be made, and what the current state of the technology is. To do this, it examines the successes and failures of various verifiable voting systems used in real-world elections.

Chapter 3 covers the cryptography that makes Helios a cryptographically end-to-end verifiable system. Topics include finite field cryptography, the ElGamal cryptosystem, underlying security assumptions, and zero-knowledge proofs of knowledge and how all of them are used in Helios. Additionally, this section contains information related to the system components, architecture, and use cases of the system.

Chapter 4 demonstrates attacks against Helios's cryptographic implementation and thus some of its cryptographic properties. This chapter contains two attacks; the poison ballot attack and the vote stealing attack by explaining their vulnerabilities, exploits, implementations, and fixes.

Chapter 5 discusses the existence of a cross-site scripting vulnerability and how even this seemingly small bug can be leveraged to mislead voters and steal their ballots.

Chapter 6 discusses the implications of the work, and what considerations need to be made moving forward when developing electronic voting systems.

Chapter 2

Background

2.1 Electronic Voting

Electronic voting is a broad collection of technologies used to assist with the various challenges of conducting an election. For each of the many tasks involved with conducting an election, there likely is an electronic method available to automate it. These tasks can be on the periphery of the election process such as voter registration; directly in the voter booth such as casting and tallying ballots; or responsible for the whole process from start to finish. However, simply automating any or all of these steps is not enough; any election body must ensure security measures present in a physical implementation transfer to an electronic one which is not trivial. The design and security of electronic voting systems is the ongoing focus for many security researchers as countries look to introduce electronic voting systems into their voting processes.

In this section, we provide an overview of various topics related to the security of electronic and internet voting systems.

2.2 Security Properties of Elections

The conveniences afforded to us by internet-connected technologies are nearly uncountable. For example, although it may be difficult, remember (or imagine) having to convert every e-mail into a letter or phone call and any online bank transfer to a visit to the actual branch. These things are rarely required now and it is not surprising that many wonder when the conveniences afforded to other activities will find their way to voting. Unfortunately, and possibly surprising to many, securing voting is not as easy as securing these other activities and the reason lies in the complex security requirements for voting systems. This section proceeds by examining

the security properties of an ideal voting system before discussing how they interact with each other.

Voter Privacy. Voter privacy, frequently provided through the *secret ballot*, is one of the most identifiable properties of free and open democracies dating back to ancient Greece [76] and represented in Article 21.3 of the United Nation's Universal Declaration of Human Rights [102]. The protection of voter privacy, primarily prevents abuse by coercing agents and by voters themselves. To guarantee this privacy, a voting system should not provide any mechanism which allows a voter to reveal their intent beyond their word.

Voter coercion is the application of force or use of intimidation by one party to influence the voting choices of another. Without any mechanism to prove voting intent, the only assurance a coercer has is what the potential victim reports. Without evidence, the perpetrator assumes all of the legal risk associated with voter coercion but has no assurance of the outcome. This imbalance is thought to have drastically reduced the prevalence of voter coercion [100].

What if the voter does not wish to exercise their right to vote but sells it instead? Although not as clearly defined legally, opponents of the practice argue it provides an unfair economic influence on the outcome by purchasing votes of the disenfranchised or poor. What the secret ballot does here is similar to the case of coercion except through increased financial risk instead of legal: by making it impossible to prove how one voted, the value of a purchased vote decreases, collapsing any vote market [59].

Voter Verifiability. In current voting systems, after the ballot is placed in the box, it is then beyond the voter's ability to ensure the rest of the process is correctly carried out. For example, if a ballot box were replaced, how would any voter be able to file a legitimate claim that such an action had taken place? The answer is she cannot in the current voting paradigm. In order to protect voter privacy, voters must give up the ability to ensure their ballot was counted.

Voting verifiability, focuses on this concern by aiming to include voters in all steps of the election. From the casting of their ballot to the counting of all ballots. By doing this, trust is shifted away from a relatively small number of people and put in the correctness of the process. The field of end-to-end verifiable voting concerns itself with development of these systems and formally defines notions of verifiability as discussed in Section 2.4.

Usability. In their seminal paper, "Why Johnny Can't Encrypt", Whitten and Tygar [105] explained the threats poor user interfaces can have on secure software. As a result of theirs and follow up works, usability has been seen as one of the largest "non-technical" threats to

system security. It has been demonstrated time and time again [85, 97, 98] that when users are asked to complete secure tasks on poorly designed systems, the usability will undermine the best-intentioned security efforts. Additionally, voting is not like sending an encrypted e-mail. Each voter usually gets one chance to perform the desired action, should it go incorrectly, it may not be until the results are released that they realize they used the system incorrectly. We can look to Palm Beach County, Florida where, during the 2000 U.S. Presidential Election between George W. Bush and Al Gore, a new ballot design caused an estimated 2000 voters to mistakenly vote for Pat Buchanan instead of Al Gore [104]. This estimate is nearly four times larger than the margin of 537 votes [71] Bush won the state by resulting in Bush winning the U.S. Presidency. This highlights the sensitivity of the electoral process to poor design.

As we have discussed, an ideal system would provide voters mechanisms allowing them to participate beyond the voting process. This increased set of interactions requires the introduction of new interfaces, each with the potential to be gravely misunderstood. This is why any voter-facing components of a secure voting system have to be designed with strong usability considerations concerning the tasks to be performed.

Dispute Resolution. The ability to verify the correctness of election results is great but how can a voter report an error in the process without revealing their vote? Additionally, how can election officials be sure that any claim of error is valid while still protecting voter privacy? This demonstrates the need for systems that are able to resolve complaints by voters, determining the validity of the claim while also preserving ballot secrecy.

Correctness. Perhaps above all else and what the rest of these properties contribute to is that, when the result is released, not only is it correct but the voters are convinced as such. These properties all align with the goal of providing an election whose result the voters can be confident in. Not only on faith in people and systems but by having evidence that the result is correct or otherwise.

Conflicting Requirements. The biggest challenge to overcome in secure electronic voting is the conflicting nature of these requirements. Voter secrecy involves no user being able to prove how they voted, but voter verification should allow voters to confirm that their vote was correctly recorded. The fulfillment of this property is only as good as the voter's ability to act on it. This introduces a conflict between privacy and dispute resolution. Affecting all of these requirements, is the actual interface through which voters and officials interact with the respective system. Individually, these conflicts are the topics of security researchers;

the resolution of their intersection is one of the things that makes secure electronic voting so difficult.

2.3 Software Independence

The concept of software independence as it relates to the security electronic voting systems was introduced by Rivest and Wack [95] and later refined by Rivest [94]. Explicitly defined as:

> *A voting system is software-independent if an (undetected) change or error in its software cannot cause an undetectable change or error in an election outcome.*

The main principle suggests that errors, undetected or intentional, in the software of an electronic voting system should not be able to produce an undetectable error that can affect an election outcome. This does not mean that voting machine software must be guaranteed to be error free; in fact, it is likely impossible to prove such a thing or so prohibitively expensive that it may as well be considered impossible. What this notion suggests, is that systems should be designed with fail safes that themselves do not explicitly rely on the correctness of the system.

One need for systems to consider this a design goal is the high complexity of voting software and the near impossibility of having zero bugs. Additionally, unlike paper ballots where a local error may go undetected and affect only a few ballots, with software distribution, a small error can exist in an entire county or state and while the error may be benign, it may also be leveraged by attackers to influence elections at a larger scale than previously possible.

Since this is more of a design philosophy than a hard fact, the roads to software independence vary greatly from application to application, but the destination is easily identifiable and there are some recurring trends in voting systems believed to be software independent.

On the identifiability of software independent systems, Rivest suggests a thought experiment such as the following: consider a malicious adversary with the ability to arbitrarily modify or replace the voting system's software. Even with this ability, could they produce an undetected change in the results? If not, this system is likely to be software-independent.

There are common design elements in software-independent systems that lend toward this property. Unsurprisingly, they all have some kind of external reference or record which is an integral part of the security the system provides. For example, the implementation of a voter-verifiable paper audit trail could be a way to add robustness to the otherwise highly software dependent direct-record electronic voting machine. In this case, a receipt is printed out that can be inspected by a voter and serve as evidence that the voting machine malfunctioned.

However, not all techniques need to be physical. The use of a public bulletin board on which ballot trackers can be printed is one way of externalizing some record or knowledge about the votes thus far which may highlight discrepancies with what voters see on a machine versus what appears on the bulletin board.

2.4 End-to-end Verifiable Elections

End-to-end verifiable voting (E2EVV) systems, referred to as verifiable voting systems, aim to provide strong software independence by providing mechanisms which allow any voter to verify the ballot casting process from his or her vote throughout the counting of all votes. To explain this, we'll cover some of the verifiability goals (i.e., what is to be verified), the mechanisms used to achieve them, and look at current state of verifiable voting.

Generally speaking, there are three main verification objectives of E2E verifiable voting systems:

1. Any voter may verify his or her own ballot was *cast as intended*,

2. Any voter may verify any ballot was *collected as cast*, and,

3. Any voter may verify that all ballots were *tallied as collected*.

An additional component of all verifiable voting systems is the introduction of a *public ledger* (also referred to as a *public bulletin board*) on which all cast ballots are posted for the public inspection. In order to protect ballot secrecy, these systems have some mechanism through which voter intent is decoupled from ballot representation on the public ledger. However, this introduces a problem; if a ballot's public representation is obfuscated from its original intent, how can a voter be convinced that the representation of his or her vote is actually included in the final tally? This problem is the motivation behind the first verification objective of end-to-end verifiable systems; cast as intended verification.

Cast as Intended. In the voting booth, a voter must first have the ability to verify that the ballot accurately represents his or her intent before submitting the ballot. In a traditional voting system, such as the paper-based system used in Canadian federal elections, this is straightforward; before submitting the ballot, a voter can visually review their ballot to make sure the circles next to their selections are correctly marked. However, if this ballot were to be posted on a bulletin board, it would be clear to everyone who the voter selected which is far

less than ideal. Knowing that ballots have to be made public for review, verifiable voting systems separate voter intent from the ballot and one of the ways this is done through the use of encryption.

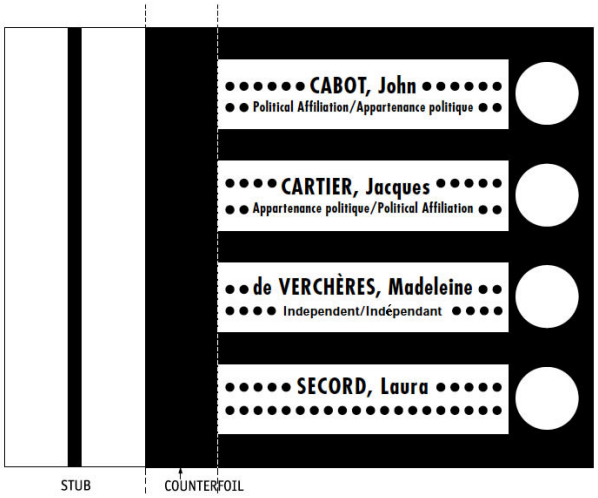

Figure 2.1: A sample ballot for elections in Canada [79].

Consider a voter named Alice. Alice wishes to cast *vote* v via *ballot* b where b is the encryption $E()$ of vote v under the election's public key, k_{pub} expressed by $b = E_{k_{pub}}(v)$. Without knowing k_{pub}, how can Alice be sure, or *verify*, that $Dec(E_{k_{pub}}(v)) = b$? This is the cast as intended verification requirement and how the requirement is satisfied depends on the voting system. In cryptographic systems, this requirement can be fulfilled with a *ballot casting assurance* protocol such as the one developed by Josh Benaloh (sometimes called the "Benaloh challenge") which proceeds as follows [60]:

- A voter at a voting machine indicates their voting preferences by marking the ballot,
- The user chooses to encrypt their ballot,
- The machine displays the encryption of their ballot and possibly the plaintext it encrypts,
- The voter is then asked if she would like to submit or audit this ballot:
  - If she chooses to submit, the ballot is digitally signed and submitted to election officials for inclusion in the election tally and the voter is provided with a receipt allowing them to look up their ballot on the bulletin board.
  - If she chooses to audit, the ballot is printed along with random factors used to generate the ciphertext.

With the plaintext, ciphertext, random factors, and knowledge of the public key, any voter can determine whether the audited ballot was encrypted properly. The voter can audit any number of ballots before choosing to cast the actual ballot preventing a malicious machine from "knowing" a challenge was coming. Additionally, only a relatively low number of voters would have to audit their ballots to provide a high level of confidence that the machines were operating correctly although more work remains to be done in quantifying the success rate of a malicious adversary against the Benaloh challenge.

Collected as Cast. After the voter is content with their audited ballot(s), she then chooses to cast her ballot and is given a receipt containing a identification number unique to the ballot. A voter can then take their receipt and compare the identification number to the public ledger to verify that their ballot has been recorded as they cast it. For this to be most effective, the public ledger needs to have a few properties mostly related to its accessibility or "how public" it is.

At this point, the voter has been assured that the ballot they submitted is a true representation of their intent, and that ballot was collected/recorded as it had been cast. All that remains is to ensure that the ballots are counted correctly.

Counted as Cast. While the nature of the previous verification steps require participation from actual voters—since they verify the correct representation of a marked plaintext ballot as it travels from the booth to the tally—counted as cast verification can be performed by anyone with access to the public ledger. With access to the ledger, anyone can validate two things:

1. All the recorded ballots have been tallied, and,

2. The finally tally is the correct summation of the encrypted ballot values.

Despite the large differences between E2EVV systems, the steps of verifying that an election was counted as cast can be generalized into a few steps. First, all the ballots need to be collected; a process not only made easier by the availability of a public bulletin board but more trustworthy as well. The easier or more public the record of votes, the more confidence voters or potential auditors can have that they are all verifying the actual tally. After collecting all the ballots, an auditing body can then begin the verification process. Unsurprisingly, this process varies from system to system but some properties to be verified include; the correctness of each ballot, the inclusion of each ballot in the tally, and the correctness of the tally itself. Cryptographic E2EVV systems such as Helios, [others], use extensive zero-knowledge proofs, and other cryptographic operations to do this allowing for fewer people to be involved in an audit

instead requiring larger amounts of computing resources. Physical systems are also subject to this kind of verification but since some of them have physical elements, more work-hours and personnel would be involved.

2.5 Categories of Internet Voting Systems

When considering the ways internet voting systems can be integrated into the larger voting process, there are three properties which determine the system's potential security as put forth by Epstein in 2011 [80]; These properties are:

1. Ballot type,
2. Type of system, and,
3. Supervision.

Each of these properties can take on two values with one lending itself to being "more" secure, creating 8 categories in total.

The first property, *ballot type*, refers to whether the system is one used for *blank ballot delivery* or *marked ballot return*. In the first case, users may receive an e-mail with a blank ballot they are expected to print off and return via a secondary channel. Marked ballot return (henceforth, ballot return), indicates some capability of the user to mark the ballot on the computer and submit it online.

The second property, *type of system*, relates to what other functions the system is expected to perform and can take on two values: *dedicated* or *non-dedicated*. Dedicated machines are only used for voting and typically are what some people see on election day and have a more localized threat model. Non-dedicated systems can include voters' mobile phones, laptops, etc.

Finally, *supervision* relates to whether or not the voting process is *supervised* by an election official or *unsupervised*. Supervised elections obviously not have less opportunity for malicious error but also accidental error in the case the voter does not correctly follow instructions.

Epstein uses these three properties to broadly categorize the security risks of any electronic voting system. With three properties and two options, we see that this ranking has eight positions with the least risky being blank ballot deliver, on dedicated system, in supervised environments. From this it follow that the riskiest configuration is a marked ballot return system on non-dedicated hardware, being used in an unsupervised environment and this is where internet voting exists.

2.6 Cryptographic Voting Systems

2.7 Hybrid End-to-End Verifiable Voting Systems

As previously mentioned, convincing the electorate that the result of an election is correct is essential to the stability of a free society and in order to truly be convinced, one must understand the argument being made. Although verifiable voting can be a complex topic, a category of systems exists that attempt to integrate well within the current voting framework. This is not only to appeal to voters but, in some cases, also leverages the benefits of the current system. These systems, which will be referred to as *hybrid end-to-end verifiable voting systems* (or *hybrid systems*) for their combination of cryptographic verifiability and paper ballots.

2.7.1 Scantegrity II

The Scantegrity II [67] (Scantegrity hereafter) voting system was designed to bring E2E verifiable voting assurances to the conventional optical-scan paper ballot used in American elections. First appearing at EVT 2008, Scantegrity aimed to provide the E2E verifiable properties previously mentioned with an addition; dispute resolution. Scantegrity was also the first end-to-end verifiable voting system to be used in a legally binding government election in the United States [65].

A lot of Scantegrity's security properties are provided through its unique paper ballot consisting of the ballot itself and a detachable receipt. The ballot portion appears similar to what one imagines when considering a ballot; a list of candidates with empty regions or "bubbles" next to their names where voters indicate their preference by filling in the empty region corresponding to their candidate. However, in Scantegrity, within each of these bubbles, printed in invisible ink, is a pseudorandomly selected alphanumeric code (also called *confirmation codes* not visible until marked with the special markers available in the voting booth. In addition to the confirmation codes, the Scantegrity ballot also contains three serial numbers designed to allow voters to identify their ballots for verification or auditing purposes.

To vote, the voter marks the bubble corresponding to her choice using the special marker in the booth. By using the marker with special ink, the area around the confirmation code turns dark first, revealing the code to the voter. Eventually, the code itself turns dark too but at a slower rate to provide the voter time to complete and submit the ballot. With the confirmation codes visible, the voter can choose to record them on the receipt or take no action. The ballot is then scanned where the voter's selections and ballot ID are recorded, and the voter is provided

with a receipt.

After polls close, voters can check audit their ballot by logging on to the election website and providing their ballot serial numbers. The website will then display the confirmation codes it understands were marked on the ballot without revealing the corresponding candidate. The voter can then compare the confirmation codes they recorded in the booth against the record.

To make verification and tallying possible, Scantegrity keeps a record of the ballot IDs, the confirmation codes, and which bubbles were selected. Through the use of various masking techniques the relationship between confirmation codes, marked bubbles, and ballot IDs is hidden to preserve voter privacy.

2.7.2 Prêt-á-Voter

Originally developed by Chaum *et al.* in 2004, Prêt-á-Voter has been improved on over time. For this section, we'll be discussing the iteration described by Ryan *et al.* in 2009. Prêt-á-Voter is a verifiable voting system that brings verifiability to the familiar paper ballot while also being accommodating to different voting systems such as Single Transferable Vote (STV).

Prêt-á-Voter's ballot is perforated down the middle into a left and right section, the left side contains a pseudorandomly permuted list of candidates and the right side contains an empty set of rows where the voter indicates their preference, by selecting the row corresponding to the candidate on the left. To vote, a voter simply marks the row corresponding to her preference, detaches the two halves, destroys the left half with the permutation, and submits the remaining half to the official for scanning. After scanning, the voter can keep her marked portion which serves as her receipt. At the bottom of the marked portion exists what Prêt-á-Voter calls a ballot cipher which is a key part in verifying the vote and voting process.

This ballot cipher serves two purposes: it provides voters with a reference to their ballot on the bulletin board allowing a voter to verify their ballot was included in the tally and it also contains (either directly or a pointer to) cryptographic information related to the permutation. Since cryptographic effort for the ballot is done before it is printed or marked, any ballot can be audited before being marked it in contrast to Benaloh's ballot casting assurance protocol preserving ballot privacy.

When looking behind the scenes of Prêt-á-Voter, we find an election authority responsible for ballot generation and mix network handling the decryption of votes. On receipt of an encrypted ballot, each node in the mix network permutes the ballot one step closer to its original representation. This prevents any group of n-1 nodes from reconstructing a ballot so long as any single node remains honest. This is a standard implementation of a mixnet as described by

Chaum in 1981 [69]. After sufficient mixing, the ballots are decrypted and tallied.

2.8 Internet Voting in the Real World

Despite all of the challenges associated with internet voting, the perceived benefits are too tempting for some governments. This section covers a few examples of elections conducted over the internet and what lessons were learned.

2.8.1 Norway

In 2011, Norway piloted the use of a verifiable remote voting system that allowed for marked ballot return as part of its local government elections and again in 2013 for its parliamentary elections.

Two companies were awarded the task of delivering an internet voting system. Ergo, a Norwegian company which had previously provided electronic voting systems to the Norwegian government; and Scytl, a Spanish company specializing in electronic voting systems.

Using a cryptographic system leveraging the homomorphic properties of ElGamal ciphertexts to provide ballot secrecy and non-interactive zero-knowledge proofs of knowledge (NIZKs) to give voters ballots were cast as intended, the companies were able to deliver a system they believed to fulfill the main requirements of correctness, secrecy, and limited coercion resistance as set out in the proposal.

The 2011 pilot included ten municipalities with approximately 160000 eligible voters between them and allowed for remote voting over the 30-day advance voting period. Of the eligible voters, approximately 28000 voters submitted their ballots electronically without any large-scale errors.

The 2013 pilot was more complicated. As technological trends moved away from Java applets, the system had to be re-written to run in voters browsers using JavaScript. While JavaScript cryptographic implementations were better developed, Scytl made an error in the pseudo-random number generator (PRNG). Namely, the reuse of a seed value for subsequent calls to the PRNG causing the first call to be sufficiently random but additional calls to be similar to previous. This resulted in approximately 60% of all ballots cast using the same random value in the encryption before the bug was patched. Fortunately, thanks to other security measures, researchers determined that no ballots were leaked and the bug was fixed [83].

2.8.2 Washington D.C.

In preparation for the 2010 midterm elections in the United States, Washington D.C.'s Board of Elections and Ethics (BOEE) piloted an internet voting system it named the D.C. Digital Vote-by-Mail Service. The system would allow overseas and military voters to cast their absentee ballots online.

Ahead of voting, voters were to receive a letter with instructions and a sixteen-character PIN that would allow them to log onto the website. The ballots were represented as PDF files that voters would download, mark using an appropriate PDF editor, and upload them to the voting server where they were to be encrypted upon receipt. After voting was complete, these ballots were to be collected on a single machine, printed, and counted among the rest of the ballots.

In a refreshing act of transparency, BOEE decided to host a mock election and invited the public to test the system's stability and security [78]. This trial started September 28, 2010 and was scheduled to end three days before the system would be made available for actual voters. Unfortunately, it only took a couple of days before a team of researchers from the University of Michigan completely compromised the voting system; gaining unlimited access not only to the system, but to the infrastructure it ran on [106]. This was made possible by poorly sanitized string handling—particularly, the use of double quotes instead of single quotes—allowing the researchers to run arbitrary commands on the server.

With this access, the researchers were able to change all of the previous votes in addition to writing software that would allow the modification of all future votes yet to be received. In addition to modifying the ballots, the team was able to read any ballot as it was received, breaking the secrecy of the vote. In addition to compromising the election, they also compromised the data center hosting the election by gaining access to the data centers surveillance cameras. It took election officials two days to realize that the intrusion had taken place even though the researchers modified the system so that a song would play after the voter submitted their ballot.

Unsurprisingly, BOEE decided not to proceed with internet voting for the 2010 midterm elections in November as the letters containing PINs could not be corrected in time and, although the vulnerability may have been small, the research team could not guarantee there were not other vulnerabilities. This case really highlights the impact seemingly small programming oversights can have in large-scale web-based voting applications.

2.8.3 New South Wales

New South Wales, a state on the East coast of Australia, conducted the largest internet voting deployment during its 2015 state election. Through the use of its iVote internet voting system, it was designed to accommodate the return of up to 280000 ballots.

Before voting, voters had to register in person, over the phone, or online. After registering, the user received an 8-digit ID and selected a 6-digit PIN. To vote online, the voter logged on to the service, authenticated themselves with their credentials, cast their ballot, and received a receipt with a 12-digit ballot tracking number. Votes were encrypted client-side, uploaded to the iVote servers, and verified by a third-party.

Until polling closed, voters could phone a verification service, provide their credentials and ballot tracking number to then have their vote read back to them. Additionally, beyond the close of polls, voters could use the ballot tracking number to verify that an associated ballot was included in the final tally although they would not receive any information about the contents of the ballot.

Shortly after the polls opened and iVote was made available to voters, a pair of researchers, J. Alex Halderman and Vanessa Teague, performed an unofficial security audit of the system. Without source code or voting credentials, they used a practice voting site set up by iVote to allow users to cast mock ballots. Although the practice site was not provided under the same pretense as it was in D.C., the researchers determined it was nearly identical to the live site and sufficient for investigation.

One of the challenges concerning the security of web applications is a common distributed architecture with many application components being delivered to the client from their respective providers. This architecture increases attack surface because not only is the main application server a vector to the client, all of the component providers are as well and that is one of the ways Halderman and Teague were able to compromise the iVote system. During their investigation, the researchers looked at the TLS configuration of the iVote server using Qualys SSL Labs's SSL Server Test:[1] an online application that provides users with a TLS "report card" for a given URL. While iVote's results came back as an effective TLS configuration, one of the providers of a third-party analytics tool, Piwik, did not do as well. Tasked with delivering JavaScript code to perform analytics, Piwik's TLS configuration scored an "F". Notably, that the server allowed 512-bit export-grade cipher suites making it vulnerable to both the FREAK [63] and Logjam [58] attacks. These downgrade attacks allowed the researchers to man-in-the-middle (MITM) the connection and replace or add code of their own desire. Being

[1] `https://www.ssllabs.com/ssltest/`

able to run arbitrary JavaScript on client machines, the researchers essentially had their choice of what to do with voters' votes; they could modify ballots, steal votes, or just break ballot secrecy. The researchers provide techniques for circumventing the verification procedures for which we refer you to the paper.

To make matters more complicated, these vulnerabilities were discovered while the polls were open making for a challenging vulnerability disclosure and need for imminent action. The vulnerability was disclosed to the Australian CERT on March 20, 2015, four days after polls opened on March 16 and was patched on March 21, 2015 after an estimate 66000 votes had been cast.

Although not as devastating a result as Washington D.C.'s Digital Vote by Mail, this case further highlights that the seemingly inherent complexity of web based applications makes them very challenging to secure to the point that they are suitable for voting.

Chapter 3

Helios Voting

3.1 Introduction

First proposed by Adida in 2008 [56], Helios is a cryptographically end-to-end verifiable internet voting system. It is designed to give anyone with a modern web browser the ability host their own election where voters can cast secret ballots, tally the votes, and verify the election every step of the way. Public facing, Helios is also designed to be easy to use and attempts to make the process of hosting or voting in a verifiable election as easy as possible.

Since its creation, Helios claims to have been used to tally over one-hundred thousand votes and has been used in various organizational elections. The Université catholique de Louvain (UCL) in Louvain-la-Neuve, Belgium, used Helios to host its presidential election in 2009 where it received ballots from nearly 4000 voters. As one of Helios's first "high-profile" elections, new considerations had to be made and large changes were made to the Helios protocol by Adida *et al.* [57] Notably, the introduction of distributed decryption and a move from mixnet-based tallying to homomorphic tallying leveraging the homomorphisms provided by the ElGamal ciphertexts already used. The cryptographic elements used in Helios are explained in further sections. Beyond its use at UCL, Helios has been used to conduct Princeton University's Undergraduate Student Government (USG) elections since 2013,[1] and the International Association for Cryptologic Research (IACR) since 2010 [82].

[1] `https://princeton.heliosvoting.org`

3.2 Helios Overview

Helios providers users and administrators with an easy-to-use interface accessed via the user's web browser. Behind this interface exists a combination of various cryptographic protocols, each of which works together to deliver a private, end-to-end verifiable election.

In line with the properties of verifiable voting systems as outlined in Section 2.4, Helios fulfills the following verifiability requirements:

Cast-as-intended verification. Helios uses a *ballot verifier* page implementing Benaloh's ballot casting assurance protocol [60] where voters can choose to cast or audit their ballot to determine if their ballots were correctly encrypted and whether the fingerprint was correctly generated.

Recorded-as-cast verification. After choosing to cast a ballot, the voter is then given a ballot fingerprint (essentially a receipt number) that is posted in the *ballot tracking center* which acts as a bulletin board containing all voter identifiers (or pseudonyms) and their corresponding ballot fingerprint.

Tallied-as-recorded verification. Helios elections are universally verifiable meaning that anyone, voter or otherwise, can access the information required to verify the correctness of the tally. To make this easier, Helios provides an *Election Verifier* page that will download the ballots, recompute the encrypted tally, and check the correctness of all the associated zero knowledge proofs in the browser. However, depending on the number of voters and contests, verification of the election tally can be computationally expensive and may be beyond the scope of what can be computed in-browser. In the case of complex ballots or large elections such as the UCL election in 2009, Helios makes the required data available for download so that it can be independently verified offline using more efficient methods.

3.3 Threat Model and Assumptions

Being a truly remote internet voting system, there are some threats that Helios cannot protect against and does not claim to such as coercion resistance or malware on the voter's computer.

Semantic security. Bernhard *et al.* [61] demonstrated that Helios is non-malleable under chosen plaintext attack (NM-CPA) and, by implication, indistinguishable under chosen plain-

text attack (IND-CPA). It should therefore be computationally infeasible for an adversary to guess a voter's intent with advantage based solely on the public audit trail.

Semi-trusted election authority. This thesis does not consider attacks in which a malicious election authority attempts to recover voting preferences from encrypted ballots—this capability is assumed. Although Helios does support multiple trustees with distributed decryption, the default and most common configuration is the single-trustee setting where the Helios server has a copy of the election private key.

Completeness and soundness. Helios produces various non-interactive zero-knowledge proofs (NIZKs) to prove that (a) a ballot is correctly formed, e.g., does not contain multiple votes, negative votes, etc., and (b) the election trustees decrypted the homomorphic tally correctly. A proof is *accepting* if a verification algorithm returns True, and *non-accepting* otherwise. We assume the Helios proofs are both *complete*, meaning (informally) that a verifier will accept valid proofs, and *sound* meaning a verifier will reject invalid proofs.

3.4 System Components

Helios itself is split into the following main components:

Election home page. On the election home page, an administrator has the ability to modify the ballot, specify the eligible voters, manage trustees, and open or close the polls. After polls are closed, the home page provides administrators with the ability to contact trustees, perform decryption, and release the election result. In addition to these functions, administrators have all the functionality of voters as well. For other users, the election home page allows registered voters to vote, non-registered voters to view the question, and allows everyone to view the Voters and Ballot Tracking Center where some verification capabilities are available.

Trustee home page. A link to this page is sent to the trustee when an administrator decides to add a trustee beyond the default Helios Server. From this page, the trustee is able to generate a key pair (or reuse one previously generated), and it is through this page that the trustee will eventually provide their decryption share to perform the final decryption.

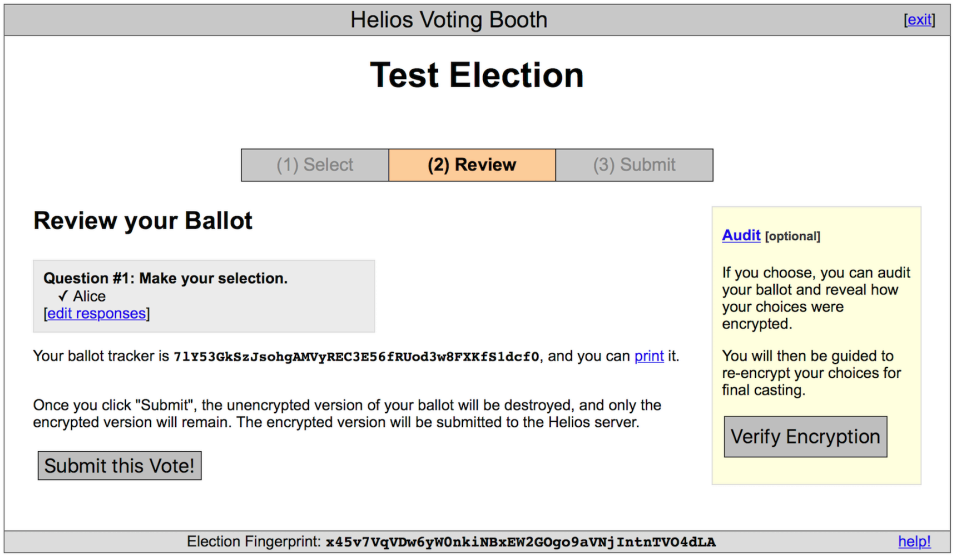

Figure 3.1: A screenshot of the Helios Voting Booth during the review stage.

Helios voting booth. The booth is responsible for nearly all ballot operations. It is where ballots are delivered, marked, and encrypted. After encryption it provides the option to audit which opens the Single Ballot Verifier.

Single ballot verifier. Accessed from the voting booth or the election home page, this verifier takes as input a JSON structure containing: a plaintext ballot, its corresponding ciphertext, random values used in the encryption, and the election ID. Using the election ID, the verifier downloads the election parameters from the server, and re-encrypts the plaintext. It then compares these values to those that were supplied for equality. Additionally, it calculates the ballot tracking ID (a hash of the ciphertext) and displays it so that a voter can independently verify it if he or she recorded it previously.

Voters and ballot tracking center. Essentially the public bulletin board for Helios, the Voters and Ballot Tracking Center contains a list of user IDs, or randomly-generated pseudonyms if the officials decide, and the corresponding Ballot Tracker ID. This is an essential part of counted as cast verification because it allows voters who recorded their ballot tracking IDs to verify their inclusion in the tally. Also available from this page is a list of audited ballots allowing non-voters to verify the encryptions as well.

Election verifier. The Election Verifier operates similarly to the Single Ballot Verifier except instead of taking a single ballot as input, it takes an election ID. Using this ID it downloads the

election parameters and all the encrypted ballots. It then recomputes the homomorphic tally, verifying the proofs along the way and outputs whether each vote and its associated proofs are valid, before determining whether the tally computation was valid.

```
{
  "answers": [
    {
      "choices": [
        {"alpha": "","beta": ""},
        {"alpha": "","beta": ""}
        ],
      "individual_proofs": [
        [{"challenge": "","commitment": {"A": "","B": ""},"response": ""},
        {"challenge": "","commitment": {"A": "","B": ""},"response": ""}],
        [{"challenge": "","commitment": {"A": "","B": ""},"response": ""},
        {"challenge": "","commitment": {"A": "","B": ""},"response": ""}]
        ],
      "overall_proof":
        [{"challenge": "","commitment": {"A": "","B": ""},"response": ""},
        {"challenge": "","commitment": {"A": "","B": ""},"response": ""}
        ]
    }
  ],
  "election_hash": "",
  "election_uuid": ""
}
```

Listing 3.1: An example of the Helios ballot for a one contest, two candidate election.

Helios ballot. The Helios ballot is represented as a JSON structure containing a list of encrypted answers, an election hash, and the election UUID. Each encrypted answer corresponds to an election question and contains an ElGamal ciphertext $\langle \alpha, \beta \rangle$ for each choice, the corresponding zero-knowledge proofs that the ciphertext encrypts a value in $[0 \ldots max]$, and an overall proof that the collection of ciphertexts is well-formed.

An example ballot for a one contest, two candidate election can be seen in Listing 3.1.

3.4.1 Process Details.

In this section we will go through the various aspects of participating in a Helios election. Starting with election setup, moving onto voting, then tallying, and finally onto verification by both voters and non-voters.

Election Setup. Helios does an excellent job of reducing the barrier to hosting online elections. The steps to hosting an election are relatively simple:

1. Log in with Facebook or Google account,
2. Provide an election name and description,
3. Add ballot questions and answers,
4. Provide a list of approved voters, or allow any registered user to vote,
5. Freeze the ballot (i.e., prevent future changes) thereby opening voting phase.

This creates an election that is publicly accessible by URL where voters are authenticated upon voting.

Setting up a key. In the event that the election administrators do not wish to trust election privacy to Helios, they can replace Helios as the default trustee. Upon doing so, the trustee receives a link to the trustee dashboard where they have to upload either their key or their key share. Either way, a trustee key also specifies domain parameters for which cryptographic operations take place and takes the form:

$$\langle g, p, q, y, x \rangle$$

defining the cyclic group $\mathbb{G}_q$ generated by g of prime order q which is a cyclic subgroup of $\mathbb{Z}_p^*$, i.e., the multiplicative group of integers modulo p. The election secret key is x and thus the election public key is $y = g^x \mod p$. More information regarding these parameters is found in Section 3.5.

Casting a ballot. This paragraph covers the actions a voter must perform in order to cast a ballot using Helios. These steps are as follows:

1. Access the election using modern browser and the election URL,
2. Click “Vote in this Election” to be taken to the Helios Voting Booth,
3. Indicate preferences (or lack of) for each question by clicking on the checkbox next to the candidate’s name. After sufficiently completing the question, the voter clicks ”Next.” If there are no remaining questions, the voter is displayed the vote review screen where they can review and change their selections or audit the ballot.

4. After choosing to submit the ballot, the user then has to authenticate (if they have not already) with one of the acceptable methods determined by Helios and the election administrators.

5. The user is then shown their "smart ballot tracker" which is the base64 representation of a `SHA256` hash of the ballot's JSON representation. The voter should to record this so they can check their ballot was recorded as cast by using the public bulletin board.

This sends the JSON representation of the ballot containing the ciphertexts for each possible choice and the associated zero-knowledge proofs to the Helios server. This representation can be recalled at any time from the ballot tracking center using their own ID.

Auditing a ballot. There are two opportunities to audit a ballot in Helios the first of which is in the voting booth where, while reviewing their ballot, the voter can choose to audit instead of submit it as per Benaloh's ballot casting assurance protocol; the second option is by viewing the "Audited Ballots" page accessible from the election homepage. The audited ballots page displays a collection of audited ballots that voters explicitly requested be posted on the publicly visible, audited ballots section. Ballots to be audited contain a standard Helios ballot that would be submitted, but also contain additional information required to conduct the audit such as the randomness used in the encryption and the actual plaintext answer. This allows anyone, registered voter or not, to be able to verify the encryption of ballots.

Either way, the process after having access to the audit tool is similar; copy the JSON from either the booth or the ballot tracking center and paste it in the single ballot verifier. The verifier begins by verifying all of the proofs of knowledge. Next, it goes through each of the plaintexts and, using its knowledge of the randomness, recomputes the ciphertexts to check if the computed ciphertext is the same as the ciphertext in the ballot. Finally, it hashes the whole ballot (without the plaintexts) and compares it to the smart ballot tracker. If any of these equalities fail, the verification fails; otherwise, the ballot verifies success.

Tallying an election. After polls close, the tallying can begin. This is the computationally intense part of the Helios process; this computation is best done outside of the browser. As a result, decryption generally occurs on the server but if an election official wished to tally the votes offline, they would likely want to use a third-party program as UCL did in 2009.

```
election_fingerprint = b64_sha(election.toJSON())

# keep track of voter fingerprints
vote_fingerprints = []

# initialize tallies to zero
tallies = [[0 for a in question.answers] for question in election.questions]

for voter in voters:
    if not verify_vote(voter.vote):
        return False

    # compute fingerprint
    vote_fingerprints.append(Base64(SHA256(voter.vote.toJSON())))

    # update tallies, looping through questions and answers within them
    for question in election.questions:
        for choice in question.answers:
            tally[question][choice] += voter.vote.answers[question].choices[choice]

# after tallying ciphertexts, verify proofs
for question in election.questions:
    for choice in question.answers:
        decryption_factor_combination = 1
        for trustee in election.trustees:
            trustee = election.trustees[trustee]
            # verify the tally for that choice within that question
            # check that it decrypts to the claimed result with the claimed proof
            if not verify_partial_decryption_proof(tallies[question][choice],
                trustee.decryption_factors[question][choice],
                trustee.decryption_proof[question][choice],
                trustee.public_key):
              return False
            # combine the decryption factors progressively
            decryption_factor_combination *= trustee.decryption_factors[question][choice]
        if (decryption_factor_combination * election.result[question][choice]) % election.public_key.p
            != tallies[question][choice].beta % election.public_key.p:
                return False
```

Listing 3.2: Pseudocode for tallying a Helios election

To tally the votes, Helios loops through all the voters and begins by verifying their overall ballot proof. If the proof verifies correctly, Helios loops through each question and option, adding the ciphertext to the tally. Modified pseudocode from the Helios documentation[2] for this process can be seen in Listing 3.2.

Auditing the election tally. Auditing the election tally is essentially the same process as tallying it again except there's now a reference against which to compare it.

3.5 Cryptography in Helios

ElGamal cryptosystem

Multiplicative group of integers modulo n An abelian group is an algebraic structure defined by a set of elements $\mathbb{G}$ and a mathematic operation $\circ$ represented together as $(\mathbb{G}, \circ)$ which observes the following properties:

- **Closure** under the group operation $\circ$ meaning that for all $a, b \in \mathbb{G}$ it is the case that $a \circ b = c \in \mathbb{G}$,
- **Associative** under the group operation $\circ$ meaning that for all $a, b, c \in \mathbb{G}$, $(a \circ b) \circ c = a \circ (b \circ c)$,
- **Commutative** under the group operation $\circ$ meaning that for all $a, b \in G$ it is the case that $a \circ b = b \circ a$
- There exists an **identity** element 1 such that, for every $a \in G$ it is the case that $a \circ 1 = a$
- The existence of an **inverse** element a^{-1} meaning for every $a \in \mathbb{G}$, there is a corresponding $a^{-1} \in \mathbb{G}$ such that $a \circ a^{-1} = 1$ where 1 is the identity element.

We can implement this definition to create the type of group related to the cryptography in Helios known as a *finite cyclic group*, specifically, *the group of multiplicative integers modulo n*.

A finite cyclic group contains a *finite* number of elements all of which which can be expressed as a repeatedly application of the group operation to a member of this group known as

[2] `http://documentation.heliosvoting.org/verification-specs/helios-v3-verification-specs`

the *generator*, denoted as g or α and still possess all of the generic group properties previously mentioned. Finite cyclic groups have many applications in cryptography but the particular group Helios's cryptography is based on is the multiplicative group of integers modulo n.

We can define the multiplicative group of integers modulo n beginning with the earlier generic group definition and definition of a finite cyclic group. From the generic definition $(\mathbb{G}, \circ)$, our set $\mathbb{G}$ will be a subgroup of the set of integers relatively prime to prime p, $\mathbb{Z}_p^* = \{1, \ldots, p-1\}$, $\mathbb{G}_q$ of prime order q where $q = \frac{p-1}{2}$ in the case p is a safe prime. This means that the subgroup contains q elements which can be produced by a special element known as the primitive root or *generator*. The generator of a group is an element that when repeatedly subjected to the group operation (such as in exponentiation), will generate all of the elements in the subgroup $\mathbb{G}_q$. The order or period of the generator is the smallest positive integer k such that

$$g^k \equiv 1 \mod p$$

where 1 is both the number and the identity element of the group.

The group generator g will be of *multiplicative order* or *period* q. In the case of the generator, order refers to

To properly define the group it stands that we need a prime p for $\mathbb{Z}_p^*$ and a generator g of prime order q to generate a subgroup $\mathbb{G}_q$. From here on, we can now express a finite cyclic group as a triplet of the group parameters, $\langle p, q, g \rangle$.

Discrete logarithm problem A discrete logarithm of an element a from a finite cyclic group defined by $\langle p, q, g \rangle$ is the positive integer k satisfying $a = g^k \mod p$. The *problem* can be defined as: given an element $a \in \mathbb{G}_q$ and a generator g of order q find k such that $a = g^k \mod p$. Currently, there is no efficient algorithm to solve the discrete logarithm problem and this difficulty is what makes it the basis for many cryptosystems. The best algorithms implemented on conventional hardware solve the DLP in $O(\sqrt{2^{\frac{n}{2}}})$ where $n = \log_2 q$ (i.e., exponential in the bit-length of the group order). However, a quantum algorithm by Shor [99] would solve the DLP in polynomial time should quantum computers come to fruition.

However, in order for the DLP to be hard, parameter selection is essential. Current NIST recommendations require $|p| \geq 2048$ bits and $|q| \geq 224$ bits, corresponding to the 112-bit security level [93].

Decisional Diffie-Hellman assumption Working in a finite cyclic group defined by $\langle p, q, g \rangle$, and given $\langle g, g^a, g^b, g^c \rangle$, for randomly selected $a, b \in \mathbb{Z}_p^*$, the DDH assumption assumes that,

without knowledge of a or b one cannot determine $c \stackrel{?}{=} ab$ with non-negligible advantage [64].

This is thought to be a greater security notion than the related *computational Diffie-Hellman assumption* which states that for a cyclic group G of order q, given $\langle g, g^a, g^b \rangle$, where g generates G_q it is computationally infeasible to produce g^{ab}.

As it relates to ElGamal and Helios, the secret information is the private key x and the ephemeral key r and thus we have

$$\langle g, g^r, g^x, g^{rx} \rangle = \langle g, \alpha, y, \beta / g^m \rangle$$

This four-tuple is referred to as a Decisional Diffie-Hellman or DDH-tuple. It will be useful in proving correctness of a ciphertext in the following sections.

ElGamal cryptosystem A public-key cryptosystem based on the Diffie-Hellman key exchange [74], the ElGamal cryptosystem [77] can be defined as a triplet of functions $\langle Gen, Enc, Dec \rangle$ respectively representing parameter generation, encryption, and decryption. For the rest of this section, we will consider users Alice and Bob who wish to secure their communication channel with ElGamal encryption.

In the parameter generation phase (*Gen*) Alice begins by generating domain parameters $\langle p, q, g \rangle$ defining a cyclic group of order q, $\mathbb{G}_q$ with primitive element or generator, g where the DDH is assumed to hold. She then randomly selects her private key, $x \in_r \mathbb{Z}_q$ and computes her public key $y = g^x \mod p$ and publishes it.

Bob now wishes to encrypt (*Enc*()) and send a message to Alice. Ciphertexts in ElGamal are a pair of group elements $\langle \alpha, \beta \rangle$. First, he obtains the domain parameters, randomly selects an ephemeral key from the group $r \in_r \mathbb{Z}_q$ and computes $\alpha = g^r \mod p$. Bob then takes Alice's public key y and uses his ephemeral key to compute the masking key $y^r = (g^x)^r = g^{xr}$. Bob then needs to encode his message m into a group element $M \in G_q$ and combines it with the masking key to create $\beta = (g^{xr}) \cdot M$. Now Bob can send the ciphertext $\langle \alpha, \beta \rangle = \langle g^r, g^{xr} \cdot M \rangle$.

For decryption (*Dec*), Alice computes the inverse of the masking key $(g^{xr})^{-1} = \alpha^{-x} = g^{-xr}$ and multiplies it with β to recover the message

$$M = \beta \cdot g^{-xr} = M \cdot g^{xr} \cdot g^{-xr}.$$

Notice that message recovery depends on the difficulty of calculating x the hardness of which is reducible to the hardness of the DLP.

In Helios, a variant of traditional ElGamal is used called *exponential ElGamal*. In it, instead of messages being encoded as a group element and then multiplied by the shared secret, the

message itself is the exponent used to generate an element (i.e., instead of $g^{xr} \cdot M$ we have $g^{xr} \cdot g^m = g^{xr+m}$). Recovery of messages in this case requires computation of the discrete logarithm of g^m however, for small message spaces, this is easily computable. By using the exponential variant, the following additive property is provided:

$$\begin{aligned} Enc(a) \otimes Enc(b) &= \langle g^{r_a} g^{r_b}, g^{a+xr_a} g^{b+xr_b} \rangle \\ &= \langle g^{r'}, g^{a+b+xr'} \rangle \\ &= Enc(a+b) \end{aligned} \tag{3.1}$$

This is known as a *homomorphism* from the field of *homomorphic encryption* which concerns itself with cryptosystems where operations on ciphertexts to produce corresponding results on the plaintexts. For the additive case in ElGamal, multiplication of ciphertexts results in the addition of their plaintext values. In the context of Helios, this allows for the tallying of encrypted ballots without requiring each ballot's decryption.

Zero knowledge proofs

A zero knowledge proof (ZKP) is a type of interactive proof system executed between two parties—henceforth named the *prover* and *verifier*—where the prover proves to the verifier that some statement is true without revealing any other information aside from the fact that the statement is true. Zero knowledge proofs must satisfy three properties:

- **Completeness.** If the prover is honest and the statement is true, the verifier will eventually be convinced of this fact,
- **Soundness.** Whether or not the prover is honest, if the statement is false, the verifier will not be convinced otherwise,
- **Zero-Knowledge.** After the proof is complete, the verifier learns nothing beyond the fact that the statement made by the prover is true.

Chaum-Pedersen proof of knowledge. The Chaum-Pedersen (CP) proof of knowledge due to Chaum and Pedersen [68] allows a prover to assert that $\langle g, g^r, g^x, g^{rx} \rangle$ forms a DDH tuple without revealing knowledge (i.e., in *zero knowledge*) of the random factor r. In the context of ElGamal, and thus Helios, we see that the DDH tuple can be represented as $\langle g, \alpha, y, \beta/g^m \rangle$. The protocol is demonstrated in Figure 3.2.

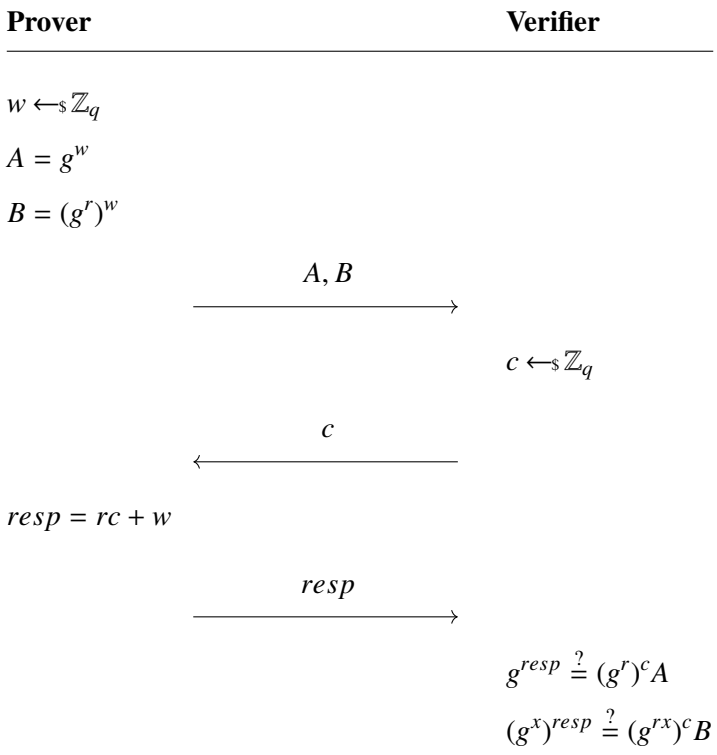

Figure 3.2: Interactive Chaum-Pedersen proof of DDH tuple $\langle g, g^r, g^x, g^{rx} \rangle$.

Fiat-Shamir Heuristic One of the limitations of interactive proof systems following the commitment, challenge, response protocol flow is that both parties to be online at the same time. However, there exists a method for turning the standard commit, challenge, response protocol flow into a one-party protocol which produces a transcript that can be verified by any observer at a later date and time.

This technique for converting interactive proof systems to non-interactive proof systems is known as the Fiat-Shamir heuristic due to Amos Fiat and Adi Shamir. It replaces the interactive "challenge" step with a non-interactive random oracle which is frequently implemented using a cryptographic hash function $H()$ producing a hash of some c = $H(context)$. Other work has shown that poorly selected context can undermine the security of the proof. Bernhard *et al.* [61] suggest methods for choosing secure contexts in this setting.

To apply this heuristic to the Chaum-Pedersen proof of knowledge in Figure 3.2, the challenge interaction is swapped with a call to the random oracle which is frequently, in the case of Helios, is a SHA256 digest of all the other commitments $A_0, B_0, \ldots, A_n, B_n$ in the proof converted to an integer mod q. With this seemingly small modification, the need for a live verifier is removed and knowledge of r has been demonstrated in zero-knowledge. This modification

can be seen in Figure 3.3.

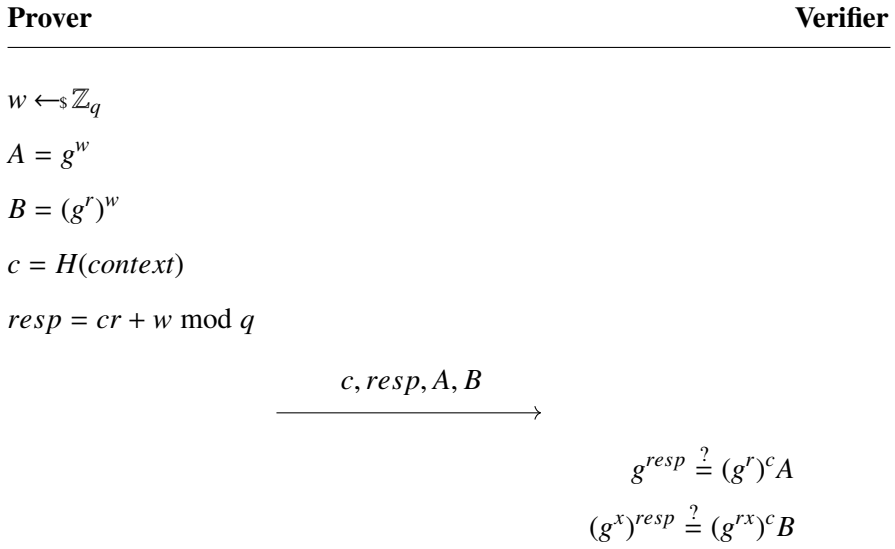

Figure 3.3: Non-interactive Chaum-Pedersen proof of DDH tuple.

Disjunctive proof system Disjunctive proofs aim to prove a logical disjunction in zero knowledge. In the context of Helios, this means to prove knowledge of only one ciphertext. Referring again to the ballot structure, where every choice has a ciphertext, there needs to be a method to prove that only $[0, \ldots, max]$ options are selected for any question to prevent over voting. The construction for this type of proof is due to Cramer *et al.* [73] and can be applied to the non-interactive Chaum-Pedersen proof. The concept at the core of disjunctive proofs is that of secret sharing: splitting up a secret and distributing it among a set of participants such that some subset of participants' shares must be combined to recover the original secret. In the case of disjunctive proofs, the secret is the `overall_challenge` and it is split across one real and one fake proofs. In the Helios ballot in Listing 3.1, notice that there are four proofs for two options (ignoring the overall proofs for now), one for each possible plaintext value. However, only two of them can be correct, otherwise, a voter could vote for multiple candidates in the same ballot. In order to do this, one of the two proofs for each candidate is simulated while the other is real.

Referring to Figure 3.3, we can see that if one knows the challenge value c in advance, it can be negated in the commitments A, B. First, as illustrated in Figure 3.4, the simulated proofs are all generated using randomly selected challenges and responses. By using the multiplicative inverse of the simulated challenge, we see that it is negated against itself in the commitment

values A, B. After the simulated proofs are completed, the real proof begins by committing to a randomly selected $w \in \mathbb{Z}_q$, then all previously commitments—simulated or otherwise—are combined and hashed together to create the overall challenge $c_{overall} = H(A_0, B_0, \ldots, A_n, B_n)$ then, all the simulated challenge values are subtracted from it leaving c_{real} the one challenge value that will be used for the real proof. By still causing the overall challenge to be based on all of the commitments before it, it leaves one degree of freedom available for the real proof to use. So, when checking the proofs, all need to verify correctly but also all of the challenge values need to sum to $c_{overall}$. If this is the case, the verifier can only know that one of the proofs was real but not be assured which one. In the case of Helios, this means that a verifier can be sure that the voter only voted once but learns this fact with zero-knowledge of which candidate the voter selected.

3.6 Related Work

A number of papers have studied the security of Helios. Estehghari and Desmedt [81] proposed an attack involving client-side malware, though the attack is admittedly outside Helios' stated threat model. Cortier and Smyth [72] identified an attack that would allow a voter to replay ballots and suggested a fix. Heiderich *et al.* [84] proposed a number of subtle client-side attacks to the web technology. Bernhard *et al.* [61] identified pitfalls in deciding what precisely to hash for the Fiat-Shamir heuristic, with implications to proof soundness. Küsters *et al.* [89] proposed the notion of clash attacks in which a corrupt election authority issues multiple voters the same receipt toward the goal of undetectably modifying the tally. Finally, Karayumak *et al.* [87] and Acemyan *et al.* [55, 54] have examined the usability of Helios and found a variety of issues, and suggested a number of places for improvement in the voter and administrator interfaces.

Simulated proof of DDH tuple for $\langle g, g^r, g^x, g^{rx} \rangle$

Prover		**Verifier**
$\mathsf{c}_{\mathsf{sim}}, \mathsf{r}_{\mathsf{sim}} \leftarrow_{\$} \mathbb{Z}_q$		
$A = g^{\mathsf{r}_{\mathsf{sim}}}(g^r)^{-c_{sim}}$		
$B = (g^x)^{\mathsf{r}_{\mathsf{sim}}}(g^{rx})^{-c_{sim}}$		
	$\xrightarrow{\mathsf{c}_{\mathsf{sim}}, \mathsf{r}_{\mathsf{sim}}, A, B}$	
		$g^{\mathsf{r}_{\mathsf{sim}}} \stackrel{?}{=} (g^r)^{c_{sim}} A$
		$(g^x)^{\mathsf{r}_{\mathsf{sim}}} \stackrel{?}{=} (g^{rx})^{c_{sim}} B$

Real proof of DDH tuple for $\langle g, g^r, g^x, g^{rx} \rangle$

Prover		**Verifier**
$w \leftarrow_{\$} \mathbb{Z}_q$		
$A = g^w$		
$B = (g^r)^w$		
$c_{overall} = H(context)$		
$\mathsf{c}_{\mathsf{real}} = \mathsf{c}_{\mathsf{overall}} - \mathsf{c}_{\mathsf{sim}} \bmod q$		
$\mathsf{r}_{\mathsf{real}} = \mathsf{c}_{\mathsf{real}} r + w \bmod q$		
	$\xrightarrow{\mathsf{c}_{\mathsf{real}}, \mathsf{r}_{\mathsf{real}}, A, B}$	
		$g^{\mathsf{r}_{\mathsf{real}}} \stackrel{?}{=} (g^r)^{c_{real}} A$
		$(g^x)^{\mathsf{r}_{\mathsf{real}}} \stackrel{?}{=} (g^{rx})^{c_{real}} B$
		$c_{overall} \stackrel{?}{=} c_{real} + c_{sim}$

Figure 3.4: Disjunctive non-interactive proofs of DDH tuple.

Chapter 4

Cryptographic Attacks

4.1 Overview

Finite-field cryptography truly is the core of Helios. In this section we look at some attacks that were made possible by some oversights regarding the underlying cryptographic properties of the groups used by the system. Consider a finite cyclic group $\mathbb{G}_q$, a subgroup of $\mathbb{Z}_p^*$, for which the discrete logarithm problem is assumed to be hard. For this to be the case, q should be prime and of sufficient length.

After visually inspecting the source code and running Helios on our own server, we realized Helios was not verifying that elements provided by users were members of $\mathbb{G}_q$. This slight oversight would allow for a malicious voter or election authority to respectively disrupt the tallying of an election or steal votes.

Implementations of these attack were carried out on a local Helios installation set up according to the installation guide in the source repository using Apache2 in place of NGINX, and Ubuntu 14.04 Desktop instead of Ubuntu 10.04. Since the attacks highlights logical flaws in the Helios source, rather than the application stack, we believe these substitutions do not have an effect on the results.

4.2 Poison Ballot Attack

In a physical voting system, the threat of election disruption is minimized by the natural distribution of polling places across a region. While some precincts may handle comparatively high numbers of ballots, the disruption of all ballot counting is nearly impossible. In internet voting, however, this risk is not inherently mitigated but instead inherently present and thus requires

careful planning and mitigation.

We discovered an incorrect assumption Helios made regarding membership of group elements resulting providing a malicious voter to craft a malformed ballot—henceforth referred to as a "poison ballot"—that would prevent an entire election from being tallied.

In a "normal" Helios election, the Helios Booth encrypts the voter's preference for each option of each question providing individual proofs for each ciphertext, and an overall proof for the complete ballot. Suppose the voter wishes to vote for Alice. An honest voting client would compute the following encryption:

$$\mathsf{Enc}_{\mathsf{Alice}}(1) = \langle g^r, g^{1+rx} \rangle \tag{4.1}$$

After the server tallies all the votes for Alice by multiplying the ciphertexts corresponding to her position on the ballot, we have a homomorphic tally of votes for Alice represented by a ciphertext $\langle \alpha_s, \beta_s \rangle$. In the first step of decryption, the election official computes $\alpha_s^x = (g^{r_s})^x$, and proves the correctness by proving $\langle g, \alpha_s, y, \alpha_s^x \rangle$ forms a DDH tuple. However, an attacker could cause this proof to fail by intentionally forcing the tuple to *not* be a DDH tuple. This can be achieved by leveraging the one thing available to every eligible voter under their complete control, the ballot.

Exploit

Recall from Equation 3.1 that a the additive homomorphic property of two ElGamal ciphertexts is achieved by taking their product. If a user were able to get a ciphertext element outside of the group included in the tally, it would result in a random value within $[1 \ldots p-1]$ with the probability of it belonging in the default, 256-bit, $\mathbb{G}_q$ being $\frac{1}{2^{1792}}$.

In order to carry out this phase of the attack, the malicious voters selects a generator h of a subgroup of order k from the prime factors of $p-1$. The Helios prime p provides us with with a number of subgroups to choose from, but for efficiency we selected h to have order $k = 2$. The malicious voter then creates the following encryption:

$$\mathsf{Enc}_{\mathsf{Alice}}(1) = \langle hg^r, g^{1+rx} \rangle \tag{4.2}$$

Accompanying every ElGamal ciphertext in the Helios ballot is a corresponding Chaum-Pedersen proof that $\langle g, \alpha, y, \beta/g^m \rangle$ forms a DDH tuple. Additionally, if the proof does not convince Helios of its validity, the ballot is invalidated and excluded from the tally.

Notice that it fails because of the challenge. Also notice, however, that if $c_{real} \equiv 0 \mod q$, that the value goes to zero. Because the challenge c_{sim} is randomly generated in the simulated

proof, this gives an attacker the ability to check whether $c_{sim} \equiv 0 \mod q$ and keep repeating the proof until receiving the appropriate challenge value.

After receiving a valid-looking proof, the attacker then submits the poison ballot and waits for the election to be tallied. Once again, let some homomorphic tally be $\langle \alpha_s, \beta_s \rangle$. The election official now runs verifies the proof, computing $\alpha_s^x = (hg^{r_s})^x$. If $x \not\equiv 0 \bmod k$ then it is easy to see the resulting

$$\langle g, \alpha_s, y, \alpha_s^x \rangle = \langle g, hg^{r_s}, hg^{xr_s} g^x \rangle,$$

is clearly not a DH tuple, meaning the verification of the decryption proof will fail. However, if $x \equiv 0 \bmod k$, the h term disappears just as it did during the proofs, and the decryption will verify. This happens with probability $1/k$, and thus k can be adjusted to make the desired outcome as likely as possible depending on the available prime factors of $p - 1$.

4.2.1 Implementation

Actual execution of this attack requires surprisingly little effort on the part of a malicious voter. Because all cryptographic operations occur on the voter's machine, the voter only needs to send the malicious value back to the server in a POST request. Additionally, because this is a web application where the cryptographic operations happen in-browser, an attacker can simply view and modify the code by using tools available in many modern browsers such as Google Developers Tools in the Google Chrome web browser.

To execute this attack, two modifications need to be made to the code. First, the `ElGamal.Ciphertext()` function in `elgamal.js`—the code responsible for ElGamal cryptographic operations in Helios—was modified. This code multiplied the original ciphertext's α value by an element belonging to a different cyclic group $\mathbb{G}_k$ generated by h of order 5. Thus the ciphertext goes from $\langle \alpha, \beta \rangle$ to $\langle \alpha h^1, \beta \rangle = \langle g^r h, g^{xr+m} \rangle$.

Next, a convincing proof is needed. In order to do this, a `do while` loop was added to the `generateDisjunctiveProof()` function to ensure that $c_{real} \equiv 0 \mod 5$. As can be seen at the bottom of Figure 3.3, a real challenge value congruent to the modified group order, effectively causes the malicious element to disappear. At this point the ballot is successfully registered and waits on the server until it is time to calculate the tally.

Let $\langle \alpha_s, \beta_s \rangle$ be the running homomorphic tally into which all votes for a particular candidate are added and that the addition operation is overloaded with ciphertext multiplication. After adding the poision ballot into the tally, the tally becomes $\langle h\alpha_s, \beta_s \rangle$. This results in a DDH tuple of $\left\langle g, h\alpha_s, y, \frac{\beta}{g^m} \right\rangle$ which is clearly not a DDH tuple causing the Chaum-Pedersen proof of the tally to fail.

4.2.2 Detection and Mitigation

In order to detect this attack, one would have to go through the previously cast ballots and verify that the ciphertext elements $\langle \alpha, \beta \rangle$ were in the group $\mathbb{G}_q$ by checking that $1 < \alpha, \beta < p - 1$ and that $\alpha^q, \beta^q \equiv 1 \mod p$ are both true. However, if not done during the initial tally, this could potentially double the amount of computation effort required to tally an election.

We worked with the Helios developers to implemented the checks above and they were merged into the master branch on May 30, 2016 they are now performed upon receipt of any ciphertext element. By checking at this point, any malformed ballot—*poison* or otherwise—gets invalidated and excluded from the tally.

4.3 Vote Stealing

In the original Helios paper [56], Adida states: "In this work, we hold the opinion that the more important property ... is unconditional integrity: even if all election administrators are corrupt, they cannot convincingly fake a tally." and this stance is understandable from the perspective of voting system design. Consider two threats: one to privacy and the other to the election tally. A threat to privacy can be thought of as a threat to the voter; they may be coerced or directly intimidated by a secondary party who can violate voter privacy protections. This threat to privacy has to act through the voters to affect an influence on the election which has inherent risks and inefficiencies. A threat to the election tally, however, supersedes a threat to privacy as it directly influences the result without the risks or inefficiencies of attempting to manipulate voters.

4.3.1 Attack Overview

This attack, the malicious adversary is an election administrator who wishes to unfairly influence or *rig* an election tally. In a physical setting, this may be modeled as a "ballot stuffing" attack where a dishonest election official may place extra ballots for the candidate they support into a ballot box. Recall that votes in Helios are encryptions of values in $[0..max]$ and although the true intent is masked by the encryption, the fact that the ballot encrypts a value in this range is assured via the accompanying proofs of knowledge; the content of which can be verified by any member of the public after the election. These proofs prevent someone, like our dishonest election administrator, from encrypting values beyond *max* and effectively *stuffing* the digital ballot box; at least, they *should*. In this section we not only discuss how a dishonest

election official may cast multiple votes from a single ballot but also *remove* (i.e., cast negative votes) from that same ballot, allowing them to arbitrarily influence the election tally all while providing convincing proofs to Helios.

4.3.2 Vulnerability

Similar to the poison ballot attack, vote stealing is made possible by insufficient checks on cryptographic elements for membership in $\mathbb{G}_q$. Although, in this case, the insufficient checks are on the domain parameters uploaded by the election trustee. This attack is thus most likely to succeed in a single-trustee election where the trustee votes in the election or the trustee has an accomplice who votes on their behalf. As per our threat model, this attack requires no modification of, or access to, Helios server components or code and could have been performed by anyone on the site.

4.3.3 Exploit

Exploiting this vulnerability begins with the election administrator submitting carefully crafted domain parameters $\langle p, q, g, y, x\rangle$ for which the expected properties apply. $|p| = 2048$, $q \mid p - 1$, $g, y \in \mathbb{G}_q$ and $y = g^x$. The only exception is we select q, and hence $|\mathbb{G}_q|$ to be as small as possible while still being large enough to accommodate all potential votes. Observing the first few factors of $p - 1$ these values can all be used to create homomorphic counters to accommodate just over 16 million votes since,

$$p - 1 = 2 \cdot 3^2 \cdot 5 \cdot 13 \cdot 23 \cdot 647 \cdot (\text{256-bit factor}) \cdot \ldots$$

Similar to the poison ballot attack, the trustee will attempt to cherry-pick challenge values to achieve their goal of submitting an arbitrary ballot with a valid proof. As an added bonus, because we're working in a small group, the trustee can decrypt the intermediate homomorphic sum of the other ballots in order to know what to encrypt to achieve the desired election result. Suppose we have an election with two voters: a honest voter, and the malicious trustee. Suppose the honest voter casts a vote for Alice:

$$\langle \mathsf{Enc}_{\mathsf{Alice}} = \mathsf{Enc}(1), \mathsf{Enc}_{\mathsf{Bob}} = \mathsf{Enc}(0)\rangle.$$

But suppose the trustee wants Bob to win. If the trustee simply casts a vote for Bob, then the result will be a tie. Instead the trustee will cast *two* votes for Bob, and *minus one* vote for Alice:

$$\langle \mathsf{Enc}_{\mathsf{Alice}} = \mathsf{Enc}(-1), \mathsf{Enc}_{\mathsf{Bob}} = \mathsf{Enc}(2)\rangle$$

such that the homomorphic tally will have the desired outcome of a landslide victory for Bob:

$$\langle \mathsf{Enc}_{\mathsf{Alice}} = \mathsf{Enc}(1 - 1 = 0), \mathsf{Enc}_{\mathsf{Bob}} = \mathsf{Enc}(0 + 2 = 2) \rangle.$$

Instead we have

$$(g^x)^{\mathsf{r}} \stackrel{?}{=} (g^{m+rx})^c B$$

and the equality does not hold. If, however, the trustee could select a challenge $c \equiv 0 \bmod q$ then we have

$$\begin{aligned} resp &= cr + w \bmod q \\ &= w \bmod q \end{aligned}$$

and therefore

$$\begin{aligned} (g^x)^{resp} &\stackrel{?}{=} (g^{m+rx})^c B \\ (g^x)^w &\stackrel{?}{=} (g^{cm+crx})(g^x)^w \\ g^{xw} &= g^{xw}. \end{aligned}$$

4.3.4 Implementation

For the parameter generation phase, we used the Python mathematics library, SageMath,[1] to partially factor $p-1$, p being the default prime used by Helios, to recover the following factors

$$p - 1 = 2 \cdot 3^2 \cdot 5 \cdot 13 \cdot 23 \cdot 647 \cdot (\text{256-bit factor}) \cdot \ldots$$

Our election parameters would then be $\langle g, y, q, p, x \rangle$ where q is one of the factors previously listed.

Initially, we tried uploading the small values for q as they were but it was of no use, they failed Helios's parameter checks and it appeared Helios had protected against such attacks. However, after further investigation, it was realized that a set of constraints, unknown at the time, were being applied to the uploaded parameters. Peculiarly, they were being checked for length and divisibility by 2. To combat this, we decided to try something new: raise one of these values to a high enough exponent to make it visually longer but still be of same order. In order for this to work, Helios would have to not check primality of q or verify the order of g which was the case. After deciding on a subgroup $\mathbb{G}_q$ of order 647 with the expanded version

[1] `http://sagemath.org`

being `q = 647`^{27} `mod p = 613295662483429012925438727699789508706335596086693371311393755083704587789 17`, the parameters were ready to be uploaded.

Back on the Helios side of things, we created an election and replaced the default Helios trustee with an account under our control. After receiving the trustee link via e-mail, we uploaded the parameters generated in the previous step and were then ready to finish setting up the election. For the election, we set up a one contest, two candidate ballot between *Option A* and *Option B*. For eligible voters, we allowed anyone who could register an account on Helios to vote in the election. In our case, we used an alternate Google account in order to register via OAuth. After freezing the ballot voting opened.

With an alternate account, we cast an honest vote for Option A. At this point, it would seem obvious that the only outcome is either a victory for Option A or a draw. At least, that would be the case in an honest election but the election official is yet to vote. Using Google Chrome's Developer Tools,[2] we modify the client-side JavaScript responsible for ElGamal cryptographic operations in two ways:

1. We put a breakpoint before the return statement in the the `ElGamal.Ciphertext()` function and,

2. Wrapped the `generateDisjunctiveProof()` function logic in a `do while` loop where it would repeat proof generation until the real challenge

```
do{...} while (c_real%647 != 0).
```

The breakpoint allows the ciphertexts based on the ballot plaintexts to be swapped with ones of the attacker's design. Remember that every selection in Helios has a corresponding ciphertext so, in the two candidate race, each voter submits two ciphertexts: an encryption of the responses for Option A and an encryption of the responses to Option B. For this attack, we swap out $\langle Enc_A(1), Enc_B(0)\rangle$ for $\langle Enc_A(-1), Enc_B(2)\rangle$.

After replacing the ciphertext, the do while loop will run until the correct challenge value generates after which the completed can be submitted. Similar to the poison ballot attack, the h term effectively disappears making the proof equality trivial and the system suspects nothing is wrong with the ciphertexts.

When computing the final tally, the proofs verify correctly and the vote-stealing ballot is included in the final tally. As can be seen in Figure 4.1, after releasing the result, if any member of the public attempts to inspect the public audit trail, they are left with a correctly verifying tally thereby breaking the soundness of Helios; an explicitly stated design goal.

[2] `https://developer.chrome.com/devtools`

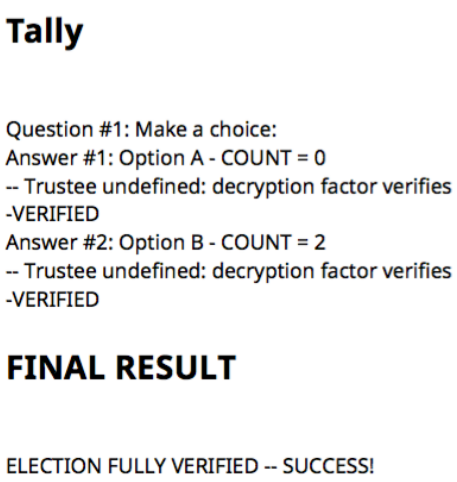

Figure 4.1: Screen capture of the Helios verifier in a rigged election. One ballot cast a vote for Option A. Another ballot cast *two* votes for Option B, and -1 votes for Option A.

4.3.5 Detection and Mitigation

As demonstrated, the attack can produce arbitrary election tallies with accepting proofs breaking the very foundation upon which Helios was built. The impact is severe since a malicious election official can not only (a) completely bypass the cryptographic protections of the cryptographic audit to produce whatever result they wish, but can also (b) produce an accepting proof that the tally was correct.

Although the tally will prove correct, the only evidence that would allow a verifier to detect that this may have occurred would be verifying that

We worked with the Helios developers and implemented the correct parameter checks to prevent this attack from happening in the future. These changes were included in the Helios master branch as of May 30, 2016[3] and ensure that the domain parameters p, q implement a cyclic group $\mathbb{G}_q$ of large prime order, and that $g, y \in \mathbb{G}_q$.

Since Helios does not provide a list of all active elections, it is difficult to verify that this attack had taken place. However, if one were to suspect that this attack had taken place in a particular election they had the link to, that person could manually verify the validity of the group parameters and would have some evidence of potential wrongdoing although it would not be conclusive without the private key. This is the role of independent verifiers for Helios and the only one that currently seems to exist—Pyrios,[4] a Helios verifier written in GO— did not provide any indication of wrongdoing when verifying our attack.

[3] `https://github.com/benadida/helios-server/commit/c1f8057ece586d0dc234c23ab57282f4f07cfd77`
[4] `https://github.com/google/pyrios`

Chapter 5

Web Attacks

5.1 Overview

Beyond the attention to detail required by the complex combination of various cryptographic functions, real-world internet voting systems have an entirely separate but equally complex threat to deal with—the environment in which they run. As discussed in Chapter 2, specifically, the elections of New South Wales and Washington D.C., sometimes the greatest threats to voting systems are not concerned with ballot encryption or decryption but what the soundness of the technology which serves the voting application. This chapter demonstrates an example of one such attacks, further highlighting the complex nature of web security and the even greater challenge of securing online voting applications.

5.2 XSS Attacks

Cross-site scripting attacks (XSS) are a type of code-injection attack where malicious code is injected into a website and then executed on a victim's machine. The malicious code often takes the form of a JavaScript element embedded in the Document Object Model (DOM). They can be difficult to completely eliminate, and may be leveraged to perform a wide range of actions on the user's behalf.

Prevention of XSS attacks generally involve sanitizing any user input by encoding scripting characters to be displayed as plain text. There is not, however, a universal solution to the problem. How user data is escaped largely depends on the context in which it will eventually be used which varies between applications. For example, user input that will be displayed as a paragraph element has different escape requirements than user input that will be assigned to a

JavaScript variable. We encountered a small oversight in the Helios code that would allow an XSS attack to be performed on a voter's device. In this section we describe an attack that would allow a remote attacker to cast a ballot on a voter's behalf, and further display the ballot had been cast as intended. Note this differs from our threat model allowing client-side malware, exploiting instead the trust the voter's browser places in Helios.

5.2.1 Vulnerability

As previously mentioned, how input will eventually be displayed affects how it should be handled. The vulnerability in Helios is caused by the use of HTML escaping on data that is used in a JavaScript context. This happens on the "Questions" page of an election where any user, registered or not, can view the questions and candidates of an election. These questions and candidates are specified by the administrator(s) of the election, and elections can be created by anyone with a Google or Facebook account.

Creating an Election. The steps for creating a Helios election are relatively simple:

1. Log in with Facebook or Google account,
2. Provide an election name and description,
3. Add ballot questions and answers,
4. Provide a list of approved voters, or allow any registered user to vote,
5. Freeze the ballot (i.e., prevent future changes) thereby opening voting phase.

After questions are added, they are serialized into JSON and stored in the Helios database. The only apparent character that is escaped is the double quote marks ("). When it comes time to serve this content to users viewing the questions page, this JSON object is retrieved from the database and parsed by Django to build the requested pages. In `election_questions.html` the JSON object is assigned to the `QUESTIONS` variable as

```
QUESTIONS = {{questions_json|safe}} .
```

The double-brace notation informs Django that a variable is contained within and will eventually replace the variable name with its value. The pipe operator informs Django that the variable is to be passed through a filter. Filters are functions that modify variables before displaying them. In the previously mentioned code, the variable, `questions_json`, is passed through the safe filter. The safe filter informs Django that the variable passed into it requires no further HTML escaping and that it is safe for display.

The consequence is that whatever an election administrator provides as a question is not escaped when displayed to other users. This is likely because HTML escaping a JSON structure results in a structure that cannot be used in JavaScript, but this also creates a perfect opportunity to perform an XSS attack.

5.2.2 Exploit

We were able to cast ballots in an election on behalf of registered voters by taking advantage of Helios's recast feature where Helios allows for the re-casting of ballots and only counts the most recently cast ballot.

To perform the attack, assume we have three parties: Alice, an honest election administrator; Bob, an honest voter; and Charlie, a remote attacker. Alice creates an election with a single question and two candidates, "Kang" and "Kodos" with open registration (i.e., any Helios user can vote) and promotes the election through the relevant channels. Alice and Bob cast their ballots through the normal process and coincidentally both vote for Kang, while Charlie casts his ballot for Kodos and wants to ensure that Bob does the same.

To do this, Charlie creates two elections: a malicious election with a ballot question containing a `<script>` tag linking to an externally hosted attack script, and another decoy election for "Favorite Soda". The malicious question exploits the XSS vulnerability can then take the form:

```
<script src=example.com/vote-stealer.js></script>
```

Charlie then sends Bob and others an email containing a link to the malicious election asking them to view his "Favorite Soda" election. While this would execute the script, the user would not necessarily be logged for the cast ballot to be accepted. To get around this, Charlie sends a link to the Google OAuth endpoint with the return parameter set to the malicious election. The victim does not necessarily detect anything unusual since the URL still points to `heliosvoting.org`. An example link would be of the form:

```
https://vote.heliosvoting.org/auth/?return_url=/helios/elections/c0 ... d9/questions
```

Users clicking on the link will first be presented with the Google login screen for Helios. After entering their credentials, the user will be redirected to the page containing the XSS payload.

We constructed an XSS payload and hosted it on a remote server. It begins by extracting a user-specific cross-site request forgery (CSRF) token from the page. Because the CSRF token is the same across all elections for the authenticated user, it is sufficient to take the CSRF

token from the malicious election page and use it in other elections. In the attack, the ballot variable is hardcoded but it could be arranged such that code from the helios-booth is used to generate ballots at runtime, although, in the interest of time, it would make more sense for the attacker to pre-generate ballots and include a new one each time the script is called. The script then builds a POST request containing the ballot and CSRF token and sends it to the Helios endpoint for Alice's election. At this point, the script has successfully cast a ballot in Alice's election from Bob's browser via the malicious election page. Upon successful completion of the POST request, the success callback is executed and the user is redirected to the "Favorite Soda" election.

From the user's point of view, they started by having to log in to Helios and were then redirected to the "Favorite Soda" election. The only evidence that a malicious action has taken place is the email confirmation for casting a ballot in Helios. Although this may alert the voter of unusual activity, Mohr *et al.* [92] recently suggested that even if voters suspect something went wrong, they may more likely attribute it to a fault or misunderstanding in their own action.

From Helios's point of view, the ballot was legitimately cast and since the most recent ballot is the one tallied, Bob's initial vote is overwritten by the ballot specified by Charlie.

5.2.3 Impact and Mitigation

We implemented the attack and were able to steal ballots from voters who clicked our malicious `heliosvoting.org` URL. The impact of this exploit is high, since anyone (not just eligible voters) can create a dummy election with the vote stealing XSS. We informed the Helios developers and they released a fix of the XSS vulnerability.

5.2.4 Related Attacks

A similar XSS vulnerability in the Helios "Questions" page was discovered in a 2011 paper by Heiderich *et al.* [84] caused by a lack of context-sensitive filtering. Heiderich is also credited on the Helios page with the disclosure[1] of a different XSS vulnerability from the paper resulting in a fix. Our work extends theirs by successfully demonstrating a complete exploit of this vulnerability and identifying the responsible code finally resulting in its fix.

[1] `http://documentation.heliosvoting.org/attacks-and-defenses`

Chapter 6

Conclusion and Future Work

This thesis covered the discovery and exploit of various vulnerabilities in cryptographically end-to-end verifiable internet voting system, Helios Voting. Helios was performing insufficient validation of user-supplied cryptographic data which allowed two major exploits described as the poison ballot attack, and vote stealing. Additionally, Helios was not properly checking user-input data related to the questions of an election, creating a vector for a cross-site scripting attack.

The poison ballot attack allowed a disgruntled voter to cast a maliciously crafted ballot containing ciphertext elements outside of the proper cyclic group. This ballot caused the proof of the homomorphic tally to fail thereby preventing the election from being tallied. With the considerable effort that can go into planning an election, it should be clear how disruptive this could be to an organization. Additionally, even if the ballots were re-examined after and the poison ballot removed, it is hard to imagine that there would be much confidence in the voting process after such an attack.

The vote stealing attack allowed a dishonest election official to upload their own intentionally-weakened cryptographic parameters when creating an election. Because Helios was not validating that the properties generated cryptographically strong mathematical structures, it created a separate channel which would allow the election official to cast multiple votes in a single ballot or even take them away. Most significantly, these vote stealing ballots would be included in the final tally where the cryptographic audit would successfully validate the soundness of the election despite it containing ballots with multiple and negative votes.

The final attack was a cross-site scripting attack made available by a lack of validation on user-supplied input for the questions of an election a user could set up. This allowed a potential attacker to inject a script hosted on a third-party server that would then run in the

victim's browser. Cross-site scripting vulnerabilities are very versatile and can be used in many malicious ways. However, with respect to Helios, we used the external script to cast ballots in different elections on behalf of the victim without the victim's knowledge.

In addition to demonstrating these attacks, we coordinated their disclosure with the Helios developers and provided fixes that are now part of the Helios source. This highlights one of the simultaneous benefits and risks of open source software; because we had access to the source, we were able to better investigate the software and provide fixes. However, just as we did, a malicious adversary could have done the same but instead of providing fixes, performed exploits. Fortunately, through correspondence with the Helios developers, we do not believe that these vulnerabilities were exploited in any previous Helios election.

Verifiable voting aims to provide robust, transparent elections by providing mechanisms allowing anyone to demonstrate correctness of their vote and of the vote counting process. It is essential in any functioning democracy that voters trust the voting process and the result it produces. However, this work demonstrates an incorrect election result with a correct verification transcript. This raises questions related to correct proofs of spurious election results in verifiable voting systems which we leave for future work.

Internet technologies have the potential to revolutionize voting the way other industries have been dramatically changed. As demonstrated, the technologies required for securely performing these elections has not yet reached maturity. Whether the "right" protocols are yet to be written or whether there is not enough economic incentive in developing robust voting software, the result is the same and a common sentiment expressed by other researchers: although internet voting is convenient and likely the future, it should not be used for anything important right now.

Bibliography

[1] Claudia Z. Acemyan, Philip Kortum, Michael D. Byrne, and Dan S. Wallach. From Error to Error: Why Voters Could not Cast a Ballot and Verify Their Vote With Helios, Prêt à Voter, and Scantegrity II. In *USENIX Journal of Election Technology and Systems*, 2015.

[2] Claudia Z. Acemyan, Philip Kortum, Michael D. Byrne, and Dan S. Wallach. Usability of voter verifiable, end-to-end voting systems: Baseline data for helios, prêt à voter, and scantegrity ii. In *USENIX EVT/WOTE*, 2015.

[3] Ben Adida. Helios: web-based open-audit voting. In *USENIX Security Symposium*, pages 335–348, 2008.

[4] Ben Adida, Olivier de Marneffe, Olivier Pereira, and Jean-Jacques Quisquater. Electing a university president using open-audit voting: Analysis of real-world use of Helios. In *EVT/WOTE*, 2009.

[5] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, et al. Imperfect forward secrecy: How Diffie-Hellman fails in practice. In *CCS*, 2015.

[6] Toke S. Aidt and Peter S. Jensen. From open to secret ballot. *Comparative Political Studies*, 50(5):555–593, 2017.

[7] Josh Benaloh. Ballot Casting Assurance via Voter-Initiated Poll Station Auditing. *Electronic Voting Technology Workshop*, pages 1–8, 2007.

[8] David Bernhard, Olivier Pereira, and Bogdan Warinschi. How not to prove yourself: Pitfalls of the Fiat-Shamir heuristic and applications to Helios. In *ASIACRYPT*, volume 7658 of *LNCS*, pages 626–643. 2012.

[9] G. E. G. Beroggi. Secure and easy internet voting. *Computer*, 41(2):52–56, 2008.

[10] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. A messy state of the union: Taming the composite state machines of TLS. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 535–552. IEEE, 2015.

[11] Dan Boneh. *The Decision Diffie-Hellman problem*, pages 48–63. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[12] Richard T Carback, David Chaum, Jeremy Clark, John Conway, Aleksander Essex, Paul S. Hernson, Travis Mayberry, Stefan Popoveniuc, Ronald L. Rivest, Emily Shen, Alan T Sherman, and Poorvi L. Vora. Scantegrity II election at Takoma Park. In *USENIX Security Symposium*, 2010.

[13] Nicholas Chang-Fong and Aleksander Essex. The cloudier side of cryptographic end-to-end verifiable voting: A security analysis of helios. In *32nd Annual Computer Security Applications Conference (ACSAC '16), CA*, 2016.

[14] David Chaum, Richard Carback, Jeremy Clark, Aleks 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*, 2008.

[15] David Chaum and Torben Pryds Pedersen. Wallet databases with observers. In *CRYPTO*, 1992.

[16] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Commun. ACM*, 24(2):84–90, February 1981.

[17] Daniel L. Chung. Distributed Helios: Defending online voting, 2015.

[18] Federal Election Commission. 2000 presidential general election results, 2000. [Online; accessed 23-March-2017].

[19] V. Cortier and B. Smyth. Attacking and fixing helios: An analysis of ballot secrecy. In *Computer Security Foundations Symposium (CSF), 2011 IEEE 24th*, pages 297–311, 2011.

[20] Ronald Cramer, Ivan Damgård, and Berry Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *CRYPTO*, 1994.

[21] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[22] Kristen Dorey, Nicholas Chang-Fong, and Aleksander Essex. Indiscreet logs: Diffie-Hellman backdoors in TLS . In *Network and Distributed System Security Symposium (NDSS '17), San Diego, CA*, 2017.

[23] K. J. Dover and Elizabeth M. Craik. *Owls to Athens: essays on classical subjects presented to Sir Kenneth Dover*. Clarendon Press, Oxford;New York;, 1990.

[24] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO 84 on Advances in Cryptology*, pages 10–18, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[25] District Of Columbia Board Of Elections and Ethics Board. District of Columbia Board of Elections and Ethics Board Announces Public Test of Digital Vote by Mail Service. (202):20001, 2010.

[26] Elections Canada. Sample ballot papers, 2014. [Online; accessed March 31, 2017].

[27] Jeremy Epstein. Internet Voting, Security, and Privacy. *William & Mary BIll of Rights Journal*, 19(4):859–906, 2011.

[28] Saghar Estehghari and Yvo Desmedt. Exploiting the client vulnerabilities in internet e-voting systems: Hacking helios 2.0 as an example. In *USENIX EVT/WOTE*, 2010.

[29] Stuart Haber, Josh Benaloh, and Shai Halevi. The helios e-voting demo for the iacr. *IACR, May*, 2010.

[30] F. Hao and P.Y.A. Ryan. *Real-world Electronic Voting: Design, Analysis and Deployment*. Series in Security, Privacy and Trust. Taylor & Francis, 2016.

[31] Mario Heiderich, Tilman Frosch, Marcus Niemietz, and Jörg Schwenk. The bug that made me president a browser- and web-security case study on helios voting. In *VoteID*, 2011.

[32] Amir Herzberg. Why Johnny can't surf (safely)? attacks and defenses for web users. *Computers & Security*, 28(1):63–71, 2009.

[33] Fatih Karayumak, Michaela Kauer, M Maina Olembo, Tobias Volk, and Melanie Volkamer. User study of the improved Helios voting system interfaces. In *Socio-Technical Aspects in Security and Trust (STAST), 2011 1st Workshop on*, pages 37–44. IEEE, 2011.

[34] Fatih Karayumak, Maina M. Olembo, Michaela Kauer, and Melanie Volkamer. Usability analysis of helios - an open source verifiable remote electronic voting system. In *USENIX EVT/WOTE*, 2011.

[35] Oksana Kulyk, Karola Marky, Stephan Neumann, and Melanie Volkamer. Introducing proxy voting to Helios. pages 98–106. IEEE, 2016.

[36] R. Küsters, T. Truderung, and A. Vogt. Clash attacks on the verifiability of e-voting systems. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 395–409, 2012.

[37] Lucie Langer, Axel Schmidt, Johannes Buchmann, and Melanie Volkamer. A taxonomy refining the security requirements for electronic voting: Analyzing Helios as a proof of concept. pages 475–480, 2010.

[38] Ülle Madise and Tarvi Martens. E-voting in Estonia 2005. the first practice of country-wide binding internet voting in the world. *Electronic voting*, 86(2006), 2006.

[39] Ester Moher, Jeremy Clark, and Aleksander Essex. Diffusion of voter responsibility: Potential failings in E2E voter receipt checking. In *USENIX Journal of Election Systems and Technology*, 2015.

[40] National Institute of Standards and Technology (NIST). *NIST Special Publication 800-57, Part 1, Revision 4. Recommendation for Key Management. Part 1: General*. 2016.

[41] Ronald L. Rivest. On the notion of "software independence" in voting systems. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 366(1881):3759–3767, 2008.

[42] Ronald L Rivest and John P Wack. On the notion of software independence in voting systems, july 2006. *Online at http://vote. nist. gov/SI-in-voting. pdf*, 2006.

[43] Avi Rubin. Security considerations for remote electronic voting over the internet. `http://avirubin.com/e-voting.security.html`.

[44] Scott Ruoti, Jeff Andersen, Daniel Zappala, and Kent Seamons. Why Johnny still, still can't encrypt: Evaluating the usability of a modern PGP client. *arXiv preprint arXiv:1510.08555*, 2015.

[45] Steve Sheng, Levi Broderick, Colleen Alison Koranda, and Jeremy J Hyland. Why Johnny still cant encrypt: Evaluating the usability of email encryption software. In *Symposium On Usable Privacy and Security*, pages 3–4, 2006.

[46] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[47] Susan Stokes, Thad Dunning, Marcelo Nazareno, and Valeria Brusco. What killed vote buying in Britain and the United States? *Brokers, Voters, and Clientelism: The Puzzle of Distributive Politics*, 2013.

[48] Vanessa Teague and J. Alex Halderman. The New South Wales ivote system: Security failures and verification flaws in a live online election. In *VoteID*, 2015.

[49] The United Nations. *Universal Declaration of Human Rights*. dec 1948.

[50] Luke Valenta, David Adrian, Antonio Sanso, Shaanan Cohney, Joshua Fried, Marcella Hastings, J Alex Halderman, and Nadia Heninger. Measuring small subgroup attacks against Diffie-Hellman. NDSS, 2017.

[51] Jonathan N Wand, Kenneth W Shotts, Jasjeet S Sekhon, Walter R Mebane Jr, Michael C Herron, and Henry E Brady. The butterfly did it: The aberrant vote for Buchanan in Palm Beach County, Florida. *American Political Science Review*, pages 793–810, 2001.

[52] Alma Whitten and J Doug Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *Usenix Security*, volume 1999, 1999.

[53] Scott Wolchok, Eric Wustrow, Dawn Isabel, and J Alex Halderman. Attacking the Washington, D.C. Internet Voting System. pages 1–18, 2010.

Curriculum Vitae

Name:	Nicholas Chang-Fong
Post-Secondary Education and Degrees:	University of Western Ontario London, ON 2011 - 2015 B.E.Sc.
Related Work Experience:	Teaching Assistant The University of Western Ontario 2015 - 2017

Publications:

Kristen Dorey, Nicholas Chang-Fong, and Aleksander Essex. Indiscreet logs: Diffie-Hellman backdoors in TLS . In *Network and Distributed System Security Symposium (NDSS '17), San Diego, CA*, 2017

Nicholas Chang-Fong and Aleksander Essex. The cloudier side of cryptographic end-to-end verifiable voting: A security analysis of helios. In *32nd Annual Computer Security Applications Conference (ACSAC '16), CA*, 2016