New Standards for E-Voting Systems: Reflections on Source Code Examinations

Thomas Haines[1] and Peter Roenne[2]

[1] Norwegian University of Science and Technology, Trondheim, Norway
thomas.haines@ntnu.no
[2] Université du Luxembourg, Luxembourg, Luxembourg
peter.roenne@uni.lu

Abstract. There is a difference between a system having no known attacks and the system being secure—as cryptographers know all too well. Once we begin talking about the implementations of systems this issue becomes even more prominent since the amount of material which needs to be scrutinised skyrockets. Historically, lack of transparency and low standards for e-voting system implementations have resulted in a culture where open source code is seen as a gold standard; however, this ignores the issue of the comprehensibility of that code.
In this work we make concrete empirical recommendations based on our, and others, experiences and findings from examining the source code of e-voting systems. We highlight that any solution used for significant elections should be well designed, carefully analysed, deftly built, accurately documented and expertly maintained. Until e-voting system implementations are clear, comprehensible, and open to public scrutiny security standards are unlikely to improve.

Keywords: Voting · Implementation · Standards

Disclaimer: This paper contains praises and critiques of the actions taken by a number of different e-voting vendors. Our evaluations are given for the purpose of clarifying what ought to be done; these examples are not intended to single out any particular vendor nor to suggest that all vendors exhibit all the issues discussed.

1 Introduction

The theoretical foundations of verifiable electronic voting are fairly mature. Simple schemes such as Helios [Adi08], using homomorphic tallying, are well-known and theoretically easy to construct. Helios and other end-to-end verifiable schemes draw on a number of techniques to make the election software-independent [Riv08]. The idea of software-independence is that system should produce evidence which shows the election result was correct regardless of any flaws in the software used during the election. However, the term is perhaps slightly misleading because there is still a fundamental reliance on the software that checks the evidence.

We are now seeing small but critical bugs in the implementation of even theoretical simple schemes. For instance, the Swiss Post system contained many components which were broken despite extensive review [HLPT20]. This is the tip of the proverbial iceberg in terms of failures and issues in allegedly end-to-end verifiable systems; other examples have included the iVote system [HT15] deployed in the Australian state of New South Wales, the e-voting system used in national elections in Estonia [SFD$^+$14], the Moscow voting system [GG20], and the issues with Voatz [SKW20] and Democracy Live [SH20].

The reason that these bugs slipped through varies but in many cases it is due to their nature. These bugs are not standard programming bugs which might be caught by standard best practice techniques. Rather, the code does not correctly capture the logical flow of the cryptographic protocol—possibly undermining the security guarantees which the cryptographic primitives were supposed to provide. Compounding these issues is that many of the bugs were present in the specification as well as the code. The bugs that we know of, were revealed when the systems were open sourced and experts in the field were able to examine the source. However, in none of these cases did the experts suggest that the list of bugs they found was exhaustive.

There is work underway to construct verifiers for e-voting schemes which are machine checked to be cryptographically secure [HGT19][HGS21]. However, these approaches cover only the publicly verifiable aspects of integrity. For deployed schemes we wish to ensure, in addition to universal verifiability, that the source code follows best practice and does not allow any obvious privacy attacks. The privacy of the deployed scheme is far more brittle than integrity and far harder to check. While we may hope that in future the code used to run elections has been machine checked to have many of the security properties we want, that day seems far off.

In this work we will discuss our, and others, experiences with examining the source code of deployed systems. We highlight problems with the processes which hampered the inspection of the systems. Based on these problems we highlight key recommendations; following these recommendations seems essential to allow the meaningful scrutiny of deployed systems.

1.1 The Problem

The aim of experts examining a system is not to find bugs but rather to ascertain that the system is reasonable free of bugs. While at present there is no issue with finding bugs the later aim—as we have noted—is rarely, if ever, achieved. In this paper we will discuss what steps vendors and election management bodies can take to enable better scrutiny of their systems.

Many of the lessons we will draw in this paper are related to those recently drawn by Haenni et al. [HDKL20]. However, the CHVote system they refer to was built from the ground up for a specific election system which is somewhat unusual; most vendors reuse their code in several different systems. The result of this is that several of the lessons we draw, while satisfied by CHVote, are not mentioned by Haenni et al.

Though we only use a single case study in the paper, the recommendations made in Section 3 are based on the authors' experiences in examining over a dozen different e-voting schemes, including:

- The iVote system used in the Australian states of New South Wales and Western Australia.
- The SwissPost system used in Switzerland.
- The CHVote system from Switzerland.
- The Helios system used by the International Association for Cryptologic Research.
- The Zeus system by GRNET.
- The Verify My Vote by the University of Surrey.
- The EVACS system used in the Australian Capital Territory.
- The STARVote system used in Travis county Texas.
- The UniCrypt system by the Bern University of Applied Sciences.
- The Verificatum Mixnet used in several systems.
- The ElectionGuard library by Microsoft.
- The vVote system used in the Australian state of Victoria.

In addition we draw upon the findings of other with respect to the following system:

- The Civitas system.
- The Norwegian system.

We will for each recommendation give an example of something that went wrong in practice when the recommendation was not followed and in many places also include a positive example. The examples are drawn partial from our own experience and partial from the experiences of others.

1.2 Outline of the Paper

By now, we have hopefully convinced the reader of the need to change the status quo if secure electronic voting is to be realised. We will in Section 2 take a case study of one particular flaw in the Scytl JavaScript ElGamal implementation. This flaw is not exploitable in the context of the system in which the JavaScript library is deployed; nevertheless, it will help us to demonstrate the gap between exploitable bugs and a system which is clearly reasonable free of bugs. In Section 3 we will provide recommendations on that concrete steps vendors and election management bodies should take before concluding in Section 4.

2 The Case Study

We will now give a case study which highlights the difficulties of examining systems. As we have already hinted, a major obstacle in examining system is the time constraints in relation to the complexity of the system; in general, *the simpler the system the better*. One common problem with deployed systems is that sub-components do not achieve clear security properties. This leaves reviewers chasing potential attacks throughout the system. In many cases, reviewers are unable to find an attack which will not be detected but given the degree of malleability the flaws in the sub-components allow, it is not possible to check that no such attack exists.

We have deliberately chosen a case study where a sub-component did not satisfy the claimed security but which did not break the overall system. We did this to emphasise the importance of clear and correct claims and documentation.

2.1 Introduction

In February 2020, we were given access to the released source code repository of the iVote system from the Australian state of New South Wales. The system is developed by the state electoral commission in collaboration with the vendor Scytl. The released documents include very little documentation and large amounts of code. In the recommendations below, we will talk about several problems with this repository but in this section we will focus on one particular flaw.

The encryption scheme ElGamal is commonly used in electronic voting protocols, including iVote, to provide privacy for the votes. ElGamal relies on a hardness assumption called the Decisional Diffie–Hellman problem which holds only for certain parameters of the system; in addition, the messages must be of the correct form (in the correct subgroup). If the parameters are chosen incorrectly or the messages are not of the correct form, then privacy of the votes is impacted.

Ensuring the messages are of the correct form can either be done at the application level or at the level of the crypto library. In response to our initial disclosures, Scytl indicated that they intend the checks to be performed at the application level; which they are. In examining the Scytl crypto library called cryptolib, we discovered incorrectly implemented checks that claim to ensure that the message was of the correct form. The result of which was that the code appeared to be making these checks without any actual security guarantees being achieved.

Scytl's claim that these checks belong at the application layer not in cryptolib seems inconsistent with the presence of such checks in cryptolib. It is problematic when reviewers find mistakes in the system and are expected to ignore them because they do not break the system in its current configuration, especially when this happens at the library level which is likely to be reused.

Summary of Details

- The method newZpGroupElement in cryptolib-js-mathematical service.js marks elements as belonging to the correct subgroup without checking.

- The method encrypt in cryptolib-js-elgamal encrypter.js checks that the message is marked as being in the correct subgroup but does not check that it is.
- These two flaws mean that, despite appearances, no checks that the messages belong to the message space are done before encrypting.
- These flaws do not appear to directly impact the privacy of iVote since the ivote-javascript-client-api follows the specification and maps the messages into the message space.

2.2 Background

iVote uses ElGamal encryption over the group of quadratic residues modulo a safe prime p. This group is of prime order q where $p = 2q + 1$. Care must be taken that both ciphertexts and messages belong to this group and not just the integers modulo p otherwise the semantic security of ElGamal will be broken.

ZpGroupElement The library stores values using a class called ZpGroupElement which has three values:

p: The modulus of the Zp subgroup to which this Zp element belongs.
q: The *order* of the Zp subgroup to which this Zp group element belongs.
value: The value of the group element.

newZpGroupElement The method newZpGroupElement in the cryptolib-js-mathematical service.js is responsible for converting a BigInteger (or JSON) into a ZpGroupElement. In the case of a BigInteger, it performs the following three checks:

- That p is positive and greater than 3.
- That q is positive and less than p.
- That the value of the element is positive and less than p.

Having completed these checks it creates a ZpGroupElement (new ZpGroupElement(p, q, value)) which is marked as belonging to the subgroup of order q.

Problem The checks suffice to ensure the value is in the multiplicative subgroup of $\mathbb{Z}_p$ but do not suffice to ensure that it belongs to the subgroup of order q.

Encrypt The method *encrypt* in cryptolib-js-elgamal encrypter.js creates the ElGamal ciphertexts. Before it does this, it runs checkEncryptionData which in turn calls validator.checkZpGroupElements passing the group as a parameter. This function, which is defined in cryptolib-js-elgamal input-validator.js says

> “@param Zpsubgroup [group] Optional parameter pertaining to a Zp subgroup. If this option is provided, the modulus and order of the element will be checked against those of the group.”

As we have noted the group is passed so the documentation says that the order of the element will be checked. The method then checks that the value q of the ZpGroupElement matches the value q of the group, but the order of the group element is not checked.

Problem When they are created, all elements are marked as belonging to the subgroup without check, which makes the subgroup check meaningless.

2.3 Case Study Recommendation

There are two distinct solutions to the highlighted flaw in the case study.

Alternative A The first solution is to remove the claims and incorrectly implemented checks from cryptolib which would align the code with the Scytl's claim that such checks belong at the application layer. At the very least the code needs to be clearly documented to reflect the fact that the checks do not provide the guarantees they claim.

Alternative B The second solution is to correctly implement the checks. For example, for a message x check that:

1. $1 \leq x \leq p$
2. $x^q \bmod p = 1$

We note that there are more efficient ways to perform the check for quadratic residues using the Legendre symbol.

Summary As we have stated, the problems above do not break the privacy of iVote because the application only calls the library on messages which are in the correct group.

The specific issues raised in this case study are related to a larger issue; the code of e-voting systems, and the applications in which they are used, are inadequately documented. Looking at the interactions of various academics with e-voting vendors, a trend can be observed from the vendors response to vulnerability discourses; they (at least from the public's perspective) retroactively define the security goals and in some cases engage in punitive measures which stifle public transparency. An important area of future improvement for e-voting vendors is to clearly document their code and specifications so that those examining system are aware of the intent.

3 Recommendations

To ensure e-voting systems are secure it is important the vendors and election management bodies engage with researches in an open, transparent and collaborative process. Below we give concrete recommendations of how this needs to be reflected in the process, documentation, and source code. We have split our recommendations into groups based on what phase of the project they correspond to. In practice most systems are developed in an agile not waterfall process but the phases will anyway serve as a convenient structure.

3.1 Design and Analysis Phase

1. Clear claims: The documentation accompanying the system should be clear about what security properties the system—and its sub-components—claim to achieve.

Clear claims are absolutely paramount to a transparent system. Claims of security which rapidly degrade into a very patchy set of trust assumptions as attacks are noticed are of little use when building secure systems.

Designing the system so that sub-components and phases of the protocol claim (and provide) clear security properties allows designers and examiners to structure their thinking. Further, it helps to ensure malicious behaviour is caught early.

Positive Example: The clear and high standards in Switzerland meant the bugs in the SwissPost system were clearly understood as failing to provide adequate security. Further, the presence of some many bugs at such a late stage revealed the inadequacy in the proceeding review processes. Switzerland has now refined its process to designing and reviewing system.

Negative Example: In contrast the lack of security requirements in many other countries such as Australia and Canada allows much weaker systems to operate.

2. Thorough documentation: The documentation—and source code comments—should be comprehensive, clear, correct and consistent.

Most of the systems include vast quantities of codes. A clear description of the protocol and intended functionality is crucial to understanding the system. One best practice for this, which was highlighted by the CHVote project, is the use of Pseudocode in the documentation.

Positive Example: The CHVote project[3] provides thorough documentation including pseudocode algorithms for its protocol. This allows programmers to more accurately implement the protocol and has allowed many examiners to notify the designers of major and minor findings.

Negative Example: In response to V. Teague's recent disclosures on mistakes

[3] https://eprint.iacr.org/2017/325.pdf

in Sigma protocols in the iVote system, the vendor responded with a long response.[4] The response relies on saying that the mistakes do not matter because of how the protocol is realised, however, the details of the protocol used in the document are missing in the public documentation.

3.2 Build Phase

3. Minimality: The source code provided should be minimal; it should contain only code related to the system under review.

This issue is very important since more code takes longer to scrutinise. In addition, the more irrelevant code contained in the code base, the harder it is to understand the intended flow. While we understand the difficult in releasing only the code that is used, rather than the entirety of the supporting libraries, it seems some progress in this direction is required to achieve security.

Negative Example: The Swiss Post system included a broken implementation of a disjunctive Schnorr proof [HLPT20] despite no such proof being needed by the system in violation of minimalism.

4. Buildable: The released source code should be easy to build. Preferable it should come with a configuration using a standard tool, such as Maven. The system should not depend on proprietary libraries which have not been released.

Examining and testing the system is best done not only by looking at the code but also by running it. In some cases the system is not buildable because it is incomplete in which case it is not possible to even look a the code.

Negative Example: The source code released for the iVote system is not buildable and indeed at certain key points the cryptographic code relies on proprietary libraries which have not been open sourced.

5. Executable: The system, once built, should be executable. The intended execution flow of the code should be clear either from the documentation or tests.

Particularly in JavaScript, methods can be heavily overloaded and do very different things based upon the type of input they receive. This makes it very hard to ascertain if the code behaves correctly on the input it is given during execution.

Negative Example: The Swiss Post system was not easily executable which significantly delayed the researchers examining the system.

[4] `https://www.scytl.com/wp-content/uploads/2019/11/Scytl_response-_VT_NSW_iVote_Oct2019.pdf`

6. Exportable: It should be possible to export test vectors into a well defined format for testing with an independent verifier.

While most systems support some form of export, many only export the information which is strictly necessary. Zero-knowledge proofs by design are incredible brittle to mistakes which makes debugging hard. We recommend systems to also include a complete export, including (at least some) intermediary values, which would form a better test vector.

Positive Example: The ElectionGuard system by Microsoft includes a reference implementation with a utility to run demo elections and export the resulting data. The resulting data is exported as a valid JSON file in a well defined format.

Negative Example: The election system used in the Estonian national election implements a custom hash function (based on SHA256) for use in its proofs. Including the challenge in the test vectors would ease the difficulties in assessing why an independent verifier was failing.

7. Consistent documentation and source: The source code and the documentation should correspond to each other.

As already elaborated, the review time is substantially increased by high levels of inconstancy in the source and documentation.

Negative Example: The ElectionGuard system's specification[5] required a group which was incompatible with the original reference implementation. We note that the original reference implementation was depreciated on the 15th of June 2020.

3.3 Maintain Phase

8. Regularly Updated: The open source variant of the system should be regularly updated so that experts can check that previous bugs are correctly fixed.

Most e-voting systems are regular updated with new features and patches. Every change may break the security of the system or fail to properly patch the intended vulnerability.

Negative Example: While Scytl has said they will fix the issue with the decryption proofs in iVote (that V. Teague discovered), at the time of writing the issue is still present in the semi-public repository. This prevents experts from checking the validity of the fix.[6]

[5] https://github.com/microsoft/electionguard

[6] A previous version of this paper erroneously claimed that an incorrect fix had been implemented in the semi-public repository; no fix has been implemented in the repository.

9. Minimal restrictions on disclosure: The restrictions on the disclosure of vulnerabilities should be minimal.

The infrequency with which e-voting systems are used means vulnerability disclosure has difference incentives than in other areas. If a system is not being used nor is about to be used than why should disclosure be limited at all? On the other hand, if an election is running, or about to run, the parties involved have–in many cases–a very limited legally mandated time-frame to challenge the election result and withholding disclosure through that periods seems ethically dubious at best.

Negative Example: Prominent experts were offered access to the iVote system in Australia under an agreement which forbade them from disclosing vulnerabilities publicly for five years; they understandable declined to sign the agreement. The agreement has since been updated to restrict disclosure for 45 days which has resulted in much greater engagement.

4 Conclusion

In this paper we have discussed how the flaws in the code and documentation of systems opened to scrutiny by e-voting vendors and election management prevents meaningful analysis. We have given a case study from the iVote system which demonstrates that scrutineers are expected to overlook flaws in the code and documentation; cryptographic code which does not meets its security claims should not be included even if it does not break the current configuration.

Based on this and other experiences we provide 9 recommendations to vendors and election management bodies which would enable better scrutiny of e-voting systems:

1. Clear claims: Make clear claims about the security of the system
2. Thorough documentation: Provide comprehensive, clear, correct and consistent documentation
3. Minimality: The source code should be minimal and only contain code relevant to the system under review
4. Buildable: The source code should be easily buildable
5. Executable: The built system should be executable
6. Exportable: It should be possible to export test vectors for independent verification
7. Consistent documentation and source: The documentation should correspond to the source code
8. Regularly Updated: The open source system should be updated regularly to reflect the fixes of previously found bugs
9. Minimal restrictions on disclosure: Avoid long vulnerability disclosure times

All of our recommendations seem obvious but are still regularly violated in practice. We hope that by providing them in a single resource, it will help to serve as a checklist for vendors and election management bodies.

Acknowledgements This work was supported by the Luxembourg National Research Fund (FNR) and the Research Council of Norway for the joint project SURCVS.

References

Adi08. Ben Adida. Helios: Web-based open-audit voting. In Paul C. van Oorschot, editor, *USENIX Security Symposium*, pages 335–348. USENIX Association, 2008.

GG20. Pierrick Gaudry and Alexander Golovnev. Breaking the encryption scheme of the moscow internet voting system. In *Financial Cryptography*, volume 12059 of *Lecture Notes in Computer Science*, pages 32–49. Springer, 2020.

HDKL20. Rolf Haenni, Eric Dubuis, Reto E. Koenig, and Philipp Locher. Chvote: Sixteen best practices and lessons learned. In *E-VOTE-ID*, volume 12455 of *Lecture Notes in Computer Science*, pages 95–111. Springer, 2020.

HGS21. Thomas Haines, Rajeev Goré, and Bhavesh Sharma. Did you mix me? Formally verifying verifiable mix nets in voting. In *2021 IEEE Symposium on Security and Privacy, SP 2021, San Jose, CA, USA, May 23-27, 2021*. IEEE, 2021.

HGT19. Thomas Haines, Rajeev Goré, and Mukesh Tiwari. Verified verifiers for verifying elections. In *ACM Conference on Computer and Communications Security*, pages 685–702. ACM, 2019.

HLPT20. Thomas Haines, Sarah Jamie Lewis, Olivier Pereira, and Vanessa Teague. How not to prove your election outcome. In Alina Oprea and Hovav Shacham, editors, *2020 IEEE Symposium on Security and Privacy, SP 2020, San Jose, CA, USA, May 17-21, 2020*, pages 784–800. IEEE, 2020.

HT15. J Alex Halderman and Vanessa Teague. The New South Wales iVote system: Security failures and verification flaws in a live online election. In *International Conference on E-Voting and Identity*, pages 35–53. Springer, 2015.

Riv08. Ronald L Rivest. On the notion of 'software independence' in voting systems. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366(1881):3759–3767, 2008.

SFD$^+$14. Drew Springall, Travis Finkenauer, Zakir Durumeric, Jason Kitcat, Harri Hursti, Margaret MacAlpine, and J Alex Halderman. Security analysis of the Estonian internet voting system. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 703–715. ACM, 2014.

SH20. M Specter and J Halderman. Security analysis of the democracy live online voting system, 2020.

SKW20. Michael A. Specter, James Koppel, and Daniel J. Weitzner. The ballot is busted before the blockchain: A security analysis of voatz, the first internet voting application used in U.S. federal elections. In *USENIX Security Symposium*, pages 1535–1553. USENIX Association, 2020.