Public review of e-voting source code: Lessons learnt from E-vote 2011*

Bjarte M. Østvold[1]†, Edvard K. Karlsen[2]

[1] Norwegian Computing Center, Oslo

[2] Norwegian University of Science and Technology, Trondheim

Abstract

In the Norwegian local elections of 2011, constituents in ten municipalities had the option to vote electronically over the Internet. The source code of the e-voting system was made publicly available by the Norwegian government, with the expressed intention to build confidence in the system and to facilitate public review of the code.

We conducted a low-effort review of this source code, finding significant problems with coding style, security, and correctness. Building on the lessons we learnt, and on general principles of good software engineering, we give recommendations to governments and others with source code where public trust is important. We end by giving specific advice to the Norwegian government on e-voting.

1 Introduction

In August 2008, the Norwegian government established the E-vote 2011 project,[1] with the intention to do limited trials with electronic voting in the 2011 local elections in Norway. Following a public tender process, the government awarded contracts to develop the E-vote 2011 source code and to perform quality assurance on it. The development contract went to EDB ErgoGroup with sub-contractor Scytl. During the 2011 municipal and county elections, voters in 10 selected municipalities had the option to vote electronically in the month leading up to election day, but not on the day itself. 27 557 voters, or 16.4% of the electorate in the 10 municipalities, deposited 55 785 ballots [15].

Before the elections the government published a version of the source code for all to read, with the following motivation:

*Full disclosure: The Norwegian Computing Center (NR) was awarded a framework agreement to do security-related consulting for the E-vote 2011 project, though NR never did any work under this agreement. NR did however do contracted work for the project on issues related to universal design; this work did not involve the authors.

†Corresponding author. Email: bjarte@nr.no. Karlsen's work was partly performed during student internships at NR.

This paper was presented at the NIK-2012 conference; see http://www.nik.no/.

[1] Official project home page, `http://www.regjeringen.no/en/dep/krd/prosjekter/e-vote-2011-project.html`

> Trust is vital for the Norwegian electoral system. It is therefore important that everyone who wishes to do so can find out how the system works. In order to safeguard this principle, the ministry is now publishing the source code for the e-voting system. In this way those who wish to do so, and who understand computer programming, can download it and inspect it.[2]
>
> [Publishing is] one of several means to make it possible for outsiders to check how the elections are carried out.[3]

We interpret these statements as saying that the Norwegian government wants to build trust in its e-voting system through a *public review* of the code, in which members of the public—ideally many of them—scrutinise the source code and form an opinion of it.

We applaud the Norwegian government for publishing source code.[4] We agree that public review of the source code is one necessary step in building trust in e-voting systems: We simply cannot trust a computer system to be part of our electoral system, if we don't know how it operates.

Our contributions are as follows:

- In order to assess the feasibility of the public review exercise, we did a limited code review of E-vote 2011. We found the source code to have significant quality problems: It is partly written in bad style and some of the code has potentially severe security and correctness issues. Our method and results appear in Section 3.
- Building on the lessons we learnt doing code review of E-vote 2011, and drawing on best-practise software engineering and general principles of open-source software development, we give recommendations to governments and others with e-voting or similar source code; this appears in Section 4.
- We provide specific analysis and advice to the Norwegian government on e-voting, in Section 5.1.

2 Previous e-voting source code reviews

In this section we survey the literature on source code reviews of e-voting systems, in order to put our own review of E-vote 2011 (Section 3) in context. The report of Jefferson et al. [12] on the SERVE system for Internet voting does not discuss the source code of the system, but instead warns about the risks of using commercial-off-the-shelf software to build e-voting systems. Kohno et al. [13] analyse a voting machine from Diebold, including the machine's C++ source code and the source code's version history. Their general remarks on the source code regard its legacy nature and the lack of comments, while their brief specific part points out a too complex function and lists comments indicative of low quality code. Das et al. [6] comment on bad programming practices in the eVACS electronic voting system, written in C; they found unsystematic checking of parameter and return values, memory leaks, duplicated code, and insecure password handling.

[2] Press release from the government in June 2011, `http://www.regjeringen.no/en/dep/krd/press/press-releases/2011/open-source-code-for-e-voting-system-on-.html`

[3] Our translation from Norwegian of text on the official blog of the E-vote 2011 project, `http://www.e-valgbloggen.no/?p=143`

[4] Note that the actual version of the source code used in the elections was not published until sometime after the conclusion of elections.

The 2007 'top-to-bottom' review of electronic voting systems certified for use in the state of California[5] included three source code reviews [3, 5, 11] of three kinds of voting machines, all of which comment on problematic issues in the source code reviewed, including security issues. Blaze et al. [3] comment on hard-coded keys, flawed implementations of cryptographic algorithms and flawed access control mechanisms, code complexity, insufficient input validation and error handling. Calandrino et al. [5] comment on lack of input validation, inconsistent secure programming, and blatant errors indicative of inexperienced programmers or non-existent code reviews. Inguva et al. [11] remark on bad privilege assignments, lack of input validation, and misuse of cryptography.

Drury et al. [7] review the security of a system, called ODBP, of voting machines connected to a central server. [6] They remark on minor quality issues with error handling and memory management.

3 Reviewing E-vote 2011

This section documents our code review of E-vote 2011. The review was a three-day effort, and our method of analysis was exploratory: We used code analysis tools—utilities which scan programs for quality problems such as buggy, frail, or overly complex code—to pinpoint problematic areas in the code, and followed up with manual analysis of the identified code.

The most significant difference between most reviews listed in Section 2 our review is one of resources and scope. Our work was not commissioned and therefore financed by the Norwegian Computing Center. Also, we did not have access to the system developers and there was little technical documentation.

In the following sections we first present the E-vote 2011 source code, then the various code analysis tools we used for our analyses, and finally the results obtained.

3.1 The E-vote 2011 source code and documentation

The source code of the E-vote 2011 project was initially released to the public in June 2011, and in October 2011 part[7] of the source was released in updated form.[8] The latter release includes the actual source code used in the elections in September of the same year; this is the release we review below. Both releases of the source code are available in a publicly readable repository.[9]

Table 1 gives basic facts about the E-vote 2011 source code. The columns in the table correspond to sub-repositories. We spent most of our efforts studying eVote-scytl due to several reasons: It appears to be the most security-critical component, containing cryptographic primitives and protocols, and it is by far the largest of the three code bases.

[5] `http://www.sos.ca.gov/voting-systems/oversight/top-to-bottom-review.htm`

[6] The C++ and Java source code ODBP was developed by Scytl, who also participated in the development of E-vote 2011. As we have not seen the source code of ODBP we do know if they share any source code.

[7] Specifically, only the sub-repositories eVote-scytl and eVote-ergo were updated in the October release.

[8] News item at the official government web-site, `http://www.regjeringen.no/nb/dep/krd/prosjekter/e-valg-2011-prosjektet/nyttomevalg/nytt-om-e-valg/2011/ny-versjon-av-kildekoden-publisert.html`

[9] `https://source.evalg.stat.no/svn/`. The repository consists of two snapshot, one for each release.

Name	Language	Lines of code
eCounting	C#	36 244
eVote-ergo	Java	78 079
eVote-scytl	Java	184 806

Table 1: Size of sub-repositories in the E-vote 2011 source code.

The E-vote 2011 project has extensive documentation describing the e-voting tender process and the system requirements: requirement specifications, contracts, tender offers, and the government's evaluation of the offers. However, the available technical documentation of the system design is presently (July 2012) rather scarce. It consists of a mathematical description of the cryptographic protocol [9] and a document listing the security goals. Notably missing is documentation of the design of the source code.

The published code base is very challenging to compile: There are unsatisfiable inter-dependencies between sub-components, files that are missing, and numerous unstated assumptions about build order and directory layout. We were only able to build parts of the code base after significant effort. These problems made it practically impossible to apply our tool selection to all of the published source code, since most of the tools operate on binaries.

The source code also includes unit and integration tests, but, because of the many build problems, it is very hard to run these tests for an external auditor.

3.2 Tools and analyses

Table 2 gives the overview of which kinds of code analysis tools where applied to which code bases of E-vote 2011.

	eVote-scytl	**eVote-ergo**	**eCounting**
General bugs	FindBugs		ReSharper
Duplicated code	PMD	PMD	
Complex code	JavaNCSS	JavaNCSS	NDepend

Table 2: Analysis tools applied to code bases.

General bugs Our primary assets are tools that search for many different kinds of bugs in programs, by analysing compiled code. We applied two such tools.

For the Java code bases, we used the open-source analysis tool FindBugs[10]. It has a database of several hundred bug patterns covering areas such as correctness, concurrency, performance, and security. FindBugs does *unsound* analysis, which means that some reported bugs may not be real bugs. However, the tool is regarded as both precise and effective by both industry practitioners and researchers [2]. We used the FindBugs 2.0 Eclipse plugin in its default configuration, except that we also enabled the 'security' bug category.

For the C# code base, we used JetBrains's ReSharper[11], a 'productivity tool' for Microsoft Visual Studio[12], that, among other features, provides code quality analysis.We ran ReSharper in its default configuration, using Visual Studio 2010.

[10] `http://findbugs.sourceforge.net/`

[11] `http://www.jetbrains.com/resharper/`

[12] `http://www.microsoft.com/visualstudio/en-us`

Duplicated code When writing code, programmers sometimes copy code from one part of a system, and reuse it elsewhere, thus creating two copies of the same code inside the system. Over time, the two copies are likely to diverge as the code is developed further. The problem with such 'copy-and-paste programming' is that when one copy is modified, for example, to fix a bug or improve the design, the other copy may also require changes, and this work is tedious and error prone.

PMD[13] is a free and open-source analysis tool targeting Java. Its 'copy-paste detector' searches for duplicated segments in a code base. We ran the detector on both Java-language code bases.

Although there are free tools for detecting duplicated C# code, we were not convinced of their quality, and therefore we ran no such detection for the C#-language code base eCounting.

Complex code Complex code is hard to understand and change, and suggest bad design decisions or sloppy programming. Cyclomatic complexity [14] is a measure of the number of different control flow paths through a basic block or procedure, and a high measure value indicates that code is risky or badly designed. We ran JavaNCSS[14] to record cyclomatic complexity for the Java code bases, and NDepend[15] to record the same metric for the eCounting code base.

3.3 Results

In this section, we first present and make general comments on the quantitative results from the various analyses. Following that, we study a selection of bugs in greater depth.

General bugs in eVote-scytl When run on the eVote-scytl code base, FindBugs reports 41 bugs with 'high' confidence and 169 bugs with 'normal' confidence. See Table 3 for a per-category overview of the bugs.

Category	High conf.	Normal conf.
Bad Practice	11	0
Correctness	16	119
Dodgy code	12	36
Multithreaded correctness	1	12
Security	1	2

Table 3: FindBugs analysis results for eVote-scytl.

General bugs in eCounting ReSharper reports more than seven thousand issues in the eCounting code base, a number that we regard as unrealistically high given that the code base consists of about 36 thousand lines of code. We therefore decided not to present a categorised overview of the ReSharper results. However, we note that more than two hundred 'code quality issues' are reported.

Duplicated code The numbers for duplicated segments found in the two Java code bases are as follows: 487 segments in eVote-scytl and 94 segments in eVote-ergo. These numbers seem high, but the majority of the reported duplication is caused by files that for questionable but not strictly unacceptable reasons exist in several locations, and most of the actual duplicated code is copied initialisation sequences and boilerplate code in tests.

[13] `http://pmd.sourceforge.net/`
[14] `http://javancss.codehaus.org/`
[15] `http://www.ndepend.com/`

Still, there are examples of what seems to be more alarming copy-paste programming. For instance, classes `DecryptingABCommand` and `DecryptingEBCommand` of package `com.scytl.evote.vsframework.client.commands.ag` share a 68-line segment. The duplicated segments starts at line 230 in class `DecryptingABCommand`, and line 259 in class `DecryptingEBCommand`.

Overly complex code The recorded average numbers for cyclomatic complexity are not alarming: eVote-scytl has average complexity 2.11, eVote-ergo has average complexity 1.85, and eCounting has average complexity 1.67. These averages also include abstract methods, but the averages for non-abstract methods alone are also relatively low. Still, there are examples of overly complex code. Within the eCounting code base, the most extreme one is the method `SetBallotNo` in class `BallotNumber` of namespace `ErgoGroup.Ecounting.PVote.Util`, which has cyclomatic complexity[16] 73, due to its deeply nested conditionals and loops (up to 8 levels deep). This method is a textbook example of bad style: incomprehensible, untestable and unmaintainable.

Example 1: Incorrect numerical comparison and misleading names

The class `ModularExponentiator` of package `com.scytl.evote.protocol.utilities` contains the following method:

```
150 private BigInteger modPowOptimized(final BigInteger base,
151   final BigInteger exponent, final BigInteger modulus) {
152   NativeBigInteger baseNative = new NativeBigInteger(base);
153
154   BigInteger factor = BigInteger.valueOf(1);
155   if (exponent.compareTo(BigInteger.valueOf(0)) == -1) {
156     factor =
157       (exponent.negate()).divide(modulus.subtract(BigInteger
158         .valueOf(1)));
159
160     BigInteger remainder =
161       (exponent.negate()).mod(modulus.subtract(BigInteger
162         .valueOf(1)));
163
164     if (!remainder.equals(0)) {
165       factor = factor.add(BigInteger.valueOf(1));
166     }
167   }
168
169   BigInteger exponentNative =
170     (factor.multiply(modulus.subtract(BigInteger.valueOf(1))))
171       .add(exponent);
172
173   BigInteger modExpNative =
174     baseNative.modPow(exponentNative, modulus);
175
176   return (new BigInteger(modExpNative.toByteArray()));
177 }
```

This method caught our eyes after FindBugs uncovered a non-obvious bug on line 164: The programmer's intention is to check whether the arbitrary-precision integer `remainder` is zero. But, comparing `remainder`, a `BigInteger`, to a primitive `int` always yields false—even when they represent the same integer. Thus, the predicate on line 164 is always true and `factor` is always modified regardless of the value of `remainder`.

After doing some quick analysis we suspect that the method is never actually invoked with negative exponents in the current code base,[17] so the bug appears to have no consequences now. It is, however, quite worrying that a bug in the cryptographic code

[16] `http://www.ndepend.com/Metrics.aspx\#ILCC`

[17] At many call sites the exponent appears to be an ElGamal private key.

exists at all—especially if this code is part of the 'cryptographic novelty' that Gjøsteen mentions [9, p. 4].

Furthermore, the method's name also struck us as unfortunate: Its containing class is named `ModularExponentiator`, and at first look the method appears to be only a faster alternative to standard-library modular exponentiation. But, further investigation suggests that the method is only to be used for *safe prime* moduli (q such that $q = 2p+1$, both q and p prime).

Example 2: Flawed random challenge generator

Class `Request` in package `com.scytl.evote.auditing.tpm.bizz` has a non-obvious bug in a procedure meant to craft a challenge for a secure message to a 'remote attestation tool'. While the code seems to provide the full randomness of `java.security.SecureRandom`, in practise the created challenge has only *1 bit of randomness*—instead of the intended 64 bits. The bug results from a failure to understand the subtleties of Java's method resolution and type coercion rules.

Here is the problematic method, called `createChallenge`:

```
99 private String createChallenge() {
100   SecureRandom sr = new SecureRandom();
101   return new Long(Math.round(sr.nextLong())).toString();
102 }
```

The name `round` is overloaded in class `java.lang.Math`. It is used for two different procedures: One accepts a `float` argument and returns `int`; the other accepts a `double` argument and returns `long`. One might assume that given a `long` argument, the `double`-accepting procedure should be invoked, after a `long`-to-`double` coercion. However, the opposite is the case: Java's specification[18] mandates that the method with the 'most specific' parameter type must be chosen; that is, the method matching the 'strictest' of all legal implicit casts. In this case it means coercing the 64-bit `long` returned from `SecureRandom.nextLong` to a 32-bit `float`, and invoking the `float`-processing `round`. In this coercion there is a many-bit loss of randomness.

Further, when a `round` method is given an argument outside the range of its return type, the method returns the extremal value closest to the argument. In this case, the input `float` has negligible chance of being representable in the `int` range, and in virtually every case the `createChallenge` procedure yields the string representation of either `Integer.MIN_VALUE` or `Integer.MAX_VALUE`.

This is a spectacular failure of secure programming. Two practices could have prevented this: Statistical testing of secure random values, and avoiding floating point numbers inside cryptographic code. Sensible secure development would be to follow both practices.

Example 3: Incorrect object comparisons

In Java basic object equality is by convention determined by the `equals` method of the root class `Object`. In their own classes, programmers override this method with specialised variants that meaningfully determine equality for the classes in question. When overriding `equals` it is also mandatory to *correctly* override the `Object` method `hashCode` [4, Item 8], which is meant for use by sets, maps and other collection classes that use hashing techniques.

[18] Java Language Specification, 3rd edition, §15.12.2.5, `http://java.sun.com/docs/books/jls/third_edition/html/expressions.html\#301183`

The method shown below, found in class `PrivateKeyData` of package `com.scytl.evote.protocol.managers.keymanager`, violates this rule. Although it has a correct `equals` method, its `hashCode` method is incorrect, and objects determined equal by `equals` may have different `hashCode` values.

```
187 public int hashCode() {
188   int hash = 7;
189
190   hash =
191     HASH_FACTOR * hash
192       + (null == getPrivateKey() ? 0 : getPrivateKey()
193           .getEncoded().hashCode());
194   hash =
195     HASH_FACTOR * hash
196       + (null == getCertificateChain() ? 0
197               : getCertificateChain().hashCode());
198
199   return hash;
200 }
```

The developers attempt to implement a sound hash function determined only by structural properties. But, the object returned by the call `getPrivateKey().getEncoded()` is a Java array, and although Java arrays have a `hashCode` method, it is only the default implementation of the Java root class `Object`, which is *not* structurally aware. The idiomatic solution is to use the structurally aware `Arrays.hashCode` method found in Java's standard library.

There are also other examples of relatively trivial correctness problems in the eVote-scytl code base—both flawed implementations of `hashCode` and `equals`, and several incorrect usages of `equals`, where objects that provably *never* can be equal are compared. One example is found in method `validateEvent` on the class `VoteValidator` in package `com.scytl.evote.evoting.vcs.audit.validator`. The equality test at line 146 of this method can never be true, and thus the bulk of this validation code is dead.

There are also examples in the eCounting code base. ReSharper uncovered the following flawed `Equals` method of class `MulitpleMarkComparer` in namespace `ErgoGroup.Ecounting.PVote.RuleEngine`.

```
149 public bool Equals(MultiMarkSelection x, MultiMarkSelection y)
150 {
151   if (Object.ReferenceEquals(x, y)) return true;
152
153   if (Object.ReferenceEquals(x, null) || Object.ReferenceEquals(y, null))
154     return false;
155
156   return x.ScanIDPK == x.ScanIDPK &&
157     x.BallotIDPK == y.BallotIDPK &&
158     x.MultiMarkOptions.FieldIndex == y.MultiMarkOptions.FieldIndex;
159 }
```

The bug is seen at line 156 where the second operand in the equality comparison should be `y.ScanIDPK` as the intention is to compare the `ScanIDPK` fields of the two objects.

3.4 Relating the results to the previous source code version

While our analyses were run against the second version of the code base, the concrete examples illustrated in our text, with the exception of Example 1, were also present in the first version of the source code, which was the version presented for public review. We also ran FindBugs on the first version of the eVote-scytl repository, and found a largely similar figure for bug numbers as we have presented for the second version.

3.5 Discussion

In this section we discuss the implications of our findings.

First, in the eVote-scytl code base, FindBugs uncovered a little over one bug per 1000 lines of code with high or normal confidence. Since these bugs were found automatically, it is likely that there are more bugs that have not been found by FindBugs. Second, the presence of duplicated and overly complex code, as well as the evidence of bad naming, is a threat to the maintainability of the code base. Third, the fact that we were able to find two errors in the cryptographic code of E-vote 2011 is alarming: This code is at the core of the system, protecting the integrity and secrecy of the election. Last, our review was a low effort exercise.

Given the importance of elections, the requirements to quality in e-voting systems must be higher than for other systems. The assessment of our code review of E-vote 2011 is: The E-vote 2011 source code, and by extensions, the system which is based on it, should not have been used in the Norwegian local elections of 2011.

4 Public review of e-voting source code

In this section we move away from the specifics of E-vote 2011 and consider the problem of conducting public reviews of e-voting source code in general.

We first discuss why a government may want to do a public review, and then give recommendations on conducting public review processes. Underlying all our recommendations is the realisation that since public reviewers are volunteers, they must be treated well and their efforts spent economically.

4.1 Motivating public review

In computer security, it is an old idea that the secrecy of a system should not depend upon the secrecy of its design, and that one benefit of openness is that more reviewers may examine the system, including its end users [16]. An e-voting system is one kind of system where public trust—in this case, the trust of the electorate—is crucial. For these 'publicly trusted' systems, organising a public review of the source code makes sense: The review can build trust with the public, and at the same time the system can benefit from work done by skilled and resourceful citizens, motivated by improving a system that they will in the future rely upon.

4.2 Essential recommendations for organisers of public reviews

Here we list essential recommendations relevant to all organisers of public reviews. We regard these as a *minimal* requirement set for effective public review of software. If these are not followed, we believe it is hard to attain the interest of any public reviewer.

- *The source code must be accompanied by technical documentation.* Reviewers have limited resources to spend; well-written and relevant documentation of the code's design saves their time.
- *The source code must be simple to build.* If reviewers cannot build the source code, they cannot run tests or write new tests, and they cannot do automatic semantic analysis of the code. Besides, buildable source code is required to use an integrated development environment to browse the code—which saves reviewer time.[19]

[19] Note that building the code does not imply being able to deploy and run the system. Such a requirement might be too strict in general, as it might require special hardware or network configuration. Still, developers must strive to make the system *testable* for reviewers.

- *The source code must be readable and idiomatic for the languages used.* If the code is complex, or uses an idiosyncratic style, then reviewers have problems understanding it, and thus they are less able find problems. Subtle parts of the code, such as complex cryptographic operations, should be commented.
- *The source code must be free of easy-to-find bugs.* If reviewers find that they are doing rudimentary quality control—a job that should have been done by the developers—or if reviewers start thinking that the code is beyond repair, then there is a risk that reviewers loose interest in the task.

These recommendations are basic software engineering advice. Following best-practise software engineering is also sound secure software development: Defects in software lead to potential exploits for attackers [10].

The E-vote 2011 source code did not follow any of the essential recommendations: The technical documentation of the design is almost non-existing or not published, a significant part of the code is not buildable, parts of the code is complex and non-idiomatic, and bugs are easily found using code analysis tools.

4.3 Additional recommendations for ambitious public reviews

We now present additional recommendations relevant only for public reviews where the organisers have strong ambitions to use the public review as a means of quality control and code improvement. In that case it is necessary to build and nurture a knowledgeable and efficient community of reviewers, and this requires a larger investment from the organisers compared with the previous kind of public review. These additional recommendations also make sense when public review is to go on for a long time, or when reviews are to be repeated several times, for example, at new releases of the code base. A review process following these recommendations resembles in many ways the peer review process conducted in open source software development, and in essence our recommendation echoes best-practice principles of open source development [1]. Still, our recommendations are mostly licence-agnostic.

- *Make it simple to report issues.* Information must flow easily and reliably from reviewers and back to the system owner, otherwise reviewer feedback may be lost. One way of doing this is to have an *issue tracker* that reviewers may use.
- *Let the review process be public.* If reviewers know what issues other reviewers have found, then they can avoid doing redundant work and also build upon the finding of others to investigate more subtle issues. Making the issue tracker mentioned above publicly readable is one step in this direction.
- *Build a reviewer community.* Building a community is about collecting and sharing information, making it easy for reviewers to find and share information with other reviewers. Possible community-building measures are fora for questions and answers, a maintained list of frequently asked questions with answers, and community-written documentation.
- *Provide incentives to individual reviewers.* One way of motivating reviewers is to provide financial incentive through a security bug bounty program. Examples of such programs are the Mozilla Bounty Program[20] and the Google's vulnerability reward programs for its web services[21] and browser[22].

[20] `http://www.mozilla.org/security/bug-bounty.html`

[21] `http://www.google.com/about/company/rewardprogram.html`

[22] `http://www.chromium.org/Home/chromium-security/vulnerability-rewards-program`

- *Let the development process be transparent.* If reviewers know how and when the code is developed, they can better organise and time their reviews, for example, reviewing only tested code or reviewing frequently changing code. Transparency is increased by having, for example, a project roadmap, public test coverage numbers and other metrics, and letting reviewers have read access to the developers' repository of source code.
- *Make reviewers do more than review.* If reviewers cross over into testers, patch committers or documentation authors they can become useful to the whole development process.

The last two recommendations affect more than the review part, and they require organising the development more similar to that of an open source project. Such an organisation may not always be possible. In particular, if reviewers submit code patches then the system owner must consider copyright issues. Also, some reviewers will be reluctant to submit code to closed-source software projects or projects protected by software patents.

The public review carried out for E-vote 2011 before the 2011 municipal elections did not appear to have ambitions in reviewer process support and community building.

5 Conclusion

Our conclusion is twofold. First, the E-vote 2011 source code does not have acceptable quality for use in an e-voting system, and therefore the E-vote 2011 system should not have been used in the Norwegian local elections of 2011.

Second, the Norwegian government's public review of the source code has been a failure. The source code used in the local elections of 2011 has several issues that would have been uncovered in a *functioning* public review of the previous software version. Instead, these issues were not identified, and thus they leaked into the version of the system used in the 2011 elections.

5.1 Advice to the Norwegian government

We offer some specific advice to the Norwegian government. First, the government should conduct an analysis of the implications of the flawed cryptographic challenge (Section 3.3) for the 2011 elections: Were any system parts or individuals potentially able to acquire privileges not intended by the system designers? For example, the tender documentation seems to indicate that the flawed challenge—with the randomness of a single coin toss—is involved in preventing replay attacks on queries to a trusted hardware module; a module that protects the integrity of core system parts [8, p. 76].

Second, the government should conduct a risk analysis of its e-voting system which takes into account the possibility of errors in the implementation, in particular, errors in the cryptographic code.

Third, the government should review its procedures for following up the software development of e-voting systems or other government systems where public trust is essential. In particular, it needs to reconsider how to do quality assurance of source code and of the processes used by the developers.

Last, if the government intends to continue developing e-voting source code, we have two specific pieces of advice. One, the government should conduct a thorough review of e-voting code before using it in elections. In particular, the E-vote 2011 source code should be thoroughly reviewed before that code is reused or developed further. Two, the

government should do effective public reviews of e-voting source code, following the recommendations given in Section 4.

References

[1] S. Androutsellis-Theotokis, D. Spinellis, M. Kechagia, and G. Gousios. Open source software: A survey from 10,000 feet. *Foundations and Trends in Technology, Information and Operations Management*, 4(3-4):187–347, 2010.

[2] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Using FindBugs on production software. In *22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, pages 805–806. ACM, 2007.

[3] M. Blaze, A. Cordero, S. Engle, C. Karlof, N. Sastry, M. Sherr, T. Stegers, and K.-P. Yee. Source code review of the sequoia voting system1, July 2007.

[4] J. Bloch. *Effective Java*. Addison-Wesley Java series. Addison-Wesley, 2008.

[5] J. A. Calandrino, A. J. Feldman, J. A. Halderman, D. Wagner, H. Yu, and W. P. Zeller. Source code review of the diebold voting system, July 2007.

[6] A. Das, Y. Niu, and T. Stegers. Security analysis of the eVACS open-source voting system, December 2005. Available at `http://goo.gl/tztiw`.

[7] D. R. Drury, D. C. Scoggins, A. M. Wilson, and D. Gustafson. Provisional qualification test report scytl release 1.0, version 1, September 2008. Published by: Florida Department of State Division of Elections.

[8] E-vote 2011/ErgoGroup/Scytl. SSA-U Appendix 2A: Contractor solution specification. E-vote 2011 document published on the project web site, December 2009. Available at `http://goo.gl/F2ElB`.

[9] K. Gjøsteen. Analysis of an internet voting protocol. Part of E-vote 2011's technical documentation, 2010. `http://eprint.iacr.org/2010/380.pdf`.

[10] G. Hoglund and G. McGraw. *Exploiting Software: How to Break Code*. Addison-Wesley, 2004.

[11] S. Inguva, E. Rescorla, H. Shacham, and D. S. Wallach. Source code review of the hart intercivic voting system, July 2007.

[12] D. Jefferson, A. D. Rubin, B. Simons, and D. Wagner. A security analysis of the secure electronic registration and voting experiment (serve), January 2004.

[13] T. Kohno, A. Stubblefield, A. D. Rubin, and D. S. Wallach. Analysis of an electronic voting system. In *IEEE Symposium on Security and Privacy*, pages 27–. IEEE Computer Society, 2004.

[14] T. J. McCabe. A complexity measure. In *Proceedings of the 2nd International Conference on Software Engineering*, ICSE '76, page 407. IEEE Computer Society Press, 1976.

[15] Office for Democratic Institutions and Human Rights. Norway: Internet voting pilot project, local government elections, 12 september 2011. OSCE/ODIHR Election Expert Team Report, March 2012.

[16] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.