Chapter 18
Functional Verification and Information Flow Analysis of an Electronic Voting System

Daniel Grahl and Christoph Scheben

Electronic voting (e-voting) systems which are used in public elections need to fulfill a broad range of strong requirements concerning both safety and security. Among those requirements are reliability, robustness, privacy of votes, coercion resistance and universal verifiability. Bugs in or manipulations of an e-voting system can have considerable influence on society. Therefore, e-voting systems are an obvious target for software verification. In addition, it makes an excellent target for a formal analysis of secure information flow. While the individual ballots must remain confidential, the public election result depends on these secrets. We will employ the precise analysis technique introduced in Chapter 13, that readily includes support for this kind of declassification.

We report on an implementation of an electronic voting system in Java. It is based on the sElect system by Küsters et al. [2011], but reduced to its essential functionality. Even though the actual components are clearly modularized, the challenge lies in the fact that we need to prove a highly nonlocal property: After all voters have cast their ballots, the server calculates the correct votes for each candidate w.r.t. the original ballots. This case study proves the preservation of privacy of votes. Altogether the considered code comprises 8 classes and 13 methods in about 150 lines of code of a rich fragment of Java. The presentation in this chapter follows the works by Scheben [2014] and Grahl [2015].

18.1 Electronic Voting

Elections form a part of everyday life that has not (yet) been fully conquered by computerized systems. This is partly due to the relatively high effort—elections do not occur often—and partly due to little public trust in e-voting security. The public discussion of this issue—in Germany at least—has revealed a high demand for secure systems and in turn a projection of high costs to construct them that lead to the introduction of electronic voting being suspended. Systems for electronic casting and tallying of votes that are in the field in other countries (e.g., the Netherlands, the

USA) are known to expose severe vulnerabilities. Apart from vote casting, computers are actually used in other activities related to elections such as voter registration or seat allocation.

A general goal is that electronic voting is at least as secure as voting with paper ballots. This includes *confidentiality* of individual votes. In particular they must not be attributable to a particular voter. But there is also an *integrity* issue: the final election result must reproduce the original voter intention; no vote must be lost, none must be manipulated. In paper-based elections, this mostly depends on trust in the election authorities and observers. In electronic voting, the idea is to issue a receipt to the voter, a so-called *audit trail*, for casting their ballot. After the votes have been tallied, the voters can then check on a public bulletin board whether their vote has actually been counted. This is called *verifiability* of the vote. To achieve verifiability and confidentiality of individual ballots/votes at the same time appears to be contradictory. The proposed solution is cryptography—that allows trails to be readable only to the voter. Some electronic voting systems also try to rule out voter *coercion* (by threatening or bribing). The idea is that trails and bulletin boards are of a form such that an attacker cannot distinguish the votes even if a coerced voter is trying to reveal his or her vote. This way, electronic voting may be even more secure than voting using paper ballots.[1] As it requires highest security guarantees, electronic voting has been frequently designated as a natural target for verification, e.g., by Clarkson, Chong, and Myers [2008].

18.2 Overview

We consider the sElect system implemented by Küsters et al., to be reduced to its essential functionality. In this system, a remote voter can cast one single vote for some candidate. This vote is sent through a secure channel to a tallying server. The secure channel is used to guarantee that voter clients are properly identified and cannot cast their vote twice. The server only publishes a result—the sum of all votes for each candidate—once all voters have cast their vote. The main modification compared to the original implementation by Küsters et al. is that messages are transmitted synchronously instead of asynchronously.

As described by Beckert et al. [2012], the goal is to show that no confidential information (i.e., votes) are leaked to the public. Obviously, the election result—a public information—does depend on confidential information. This is a desired situation. In order to allow this, the strong information flow property needs to be weakened, or parts of the confidential information need to be *declassified*. Section 13.5 shows how such a property can be formalized using Java Dynamic Logic and proven in the KeY verification system.

[1] An important practical aspect of elections is *fairness*. As argued by Bruns [2008], fairness requires a profound understanding of verifiability and confidentiality not only to security experts, but to any eligible voter. This issue is usually not considered with the present, complex systems.

Secure declassification—in the sense that parts of the secret information is purposely released (which is different from other uses of the term 'declassification' denoting the release of *any* information under certain constraints)—depends to a certain extent on functional correctness. In an election, the public result is the sum of the votes that result from secret ballots. In general, this cannot be dealt with using lightweight static analyses, such as type systems or program dependency graphs, which are still predominant in the information flow analysis world. Instead, the problem demands for semantically precise information flow analyses as provided by the direct formalization of noninterference in dynamic logic (Section 13.5).

18.2.1 Verification of Cryptographic Software

The sElect system uses cryptography and other security mechanisms. From a functional point of view, cryptography is extremely complex and it seems largely infeasible to reason about it formally. In particular, the usual assumption in cryptography that an attacker's deductive power is polynomially bounded—this is called a Dolev/Yao attacker [Dolev and Yao, 1983]—cannot be reasonably formalized. As a matter of fact, even encrypted transmission does leak information and therefore *strong secrecy* of votes—which can be expressed as noninterference—is not fulfilled: the messages sent over the network depend on the votes and could theoretically be decrypted by an adversary with unbounded computational power. As a consequence, information flow analysis techniques—like the ones presented in Section 13.5—would classify the sElect system insecure, although it is secure from a cryptographic point of view.

Küsters et al. [2011] proposed a solution to this problem: the authors showed that the real encryption of the system can be replaced by an implementation of *ideal encryption*. Ideal encryption completely decouples the sent message from the secret. Even an adversary with unbounded computational power cannot decrypt the message. The receiver can decrypt the message through some extra information sent over a secret channel which is not observable by adversaries. Küsters et al. showed that if—in the system with ideal encryption—votes do not interfere with the output to the public channel, then the system with real encryption guarantees privacy of votes. Therefore, it is sufficient to analyze the system with ideal encryption.

18.2.2 Verification Approach

Our approach combines functional verification and information flow verification, both performed with KeY. The properties are specified using the Java Modeling Language (see Chapter 7), including the extensions introduced in Section 13.4. All involved components are completely verified for their functional behavior. Additionally, the proof of confidentiality is based on a dynamic logic formalization of noninterference and theorem proving as laid out by Scheben [2014, Chapter 9]. The functional

verification lays a foundation for the confidentiality proofs as they use functional method contracts.

In order to obtain an implementation of the system that is practically verifiable, we have implemented a simplified system ourselves. In fact, we have implemented several prototypes one after another, verified each of them, and refined it (and its specification) to produce the next one. This chapter describes the final implementation of this series, see [Grahl, 2015, Chap. 9] for the complete scene.

An alternative to the above approach is outlined in Section 18.4.1. It combines functional correctness proofs in KeY with lightweight static information flow analysis as proposed by Küsters et al. [2015]. The target program is transformed in such a way that there is no declassification of information. We then prove in the KeY system that this transformation preserves the original functional behavior. This allows the static analyzer JOANA [Hammer, 2009, Graf et al., 2013]—which is sound, but incomplete—to report the absence of information flow.

18.2.3 System Description

Figure 18.1 shows a UML class diagram of the considered e-voting system. The implementation comprises, besides the clients (class `Voter`) and the server, an interface to the environment and a setup. The `main` method of the setup models the e-voting process itself. This is necessary because the security property—that privacy of votes is preserved up to the result of the election—can only be formulated with respect to a complete e-voting process rather than only the implementation of the client and the server alone. This means that we do not have a composition of distributed components, but a simulation of their interaction in a sequential program.

The basic protocol works as follows: First, voters register their respective client to the server, obtaining a unique identifier. Then, they can send their vote along with their identifier (once). Meanwhile, the server waits for a call to either receive one message (containing a voter identifier and a vote) or to close the election and post the result. In the former case, it fetches a message from the network. If the identifier is invalid (i.e., it does not belong to a registered voter) or the (uniquely identified) voter has already voted, it silently aborts the call. In any other case, the vote is counted for the respective candidate. In the latter case, the server first checks whether a sufficient condition to close the election holds,[2] and only then a result (i.e., the number of votes per candidate) is presented. This is illustrated in the sequence diagram in Figure 18.2.

This simplified representation hides many aspects essential to real systems. We assume both a working identification and that identities cannot be forged. We assume that the network does not leak any information about the ballot (i.e., voter identifier and vote). This is meant to be assured through means of cryptography. The network may leak—and probably will in practice—other information such as networking

[2] In the present implementation, this is when all voters have voted.

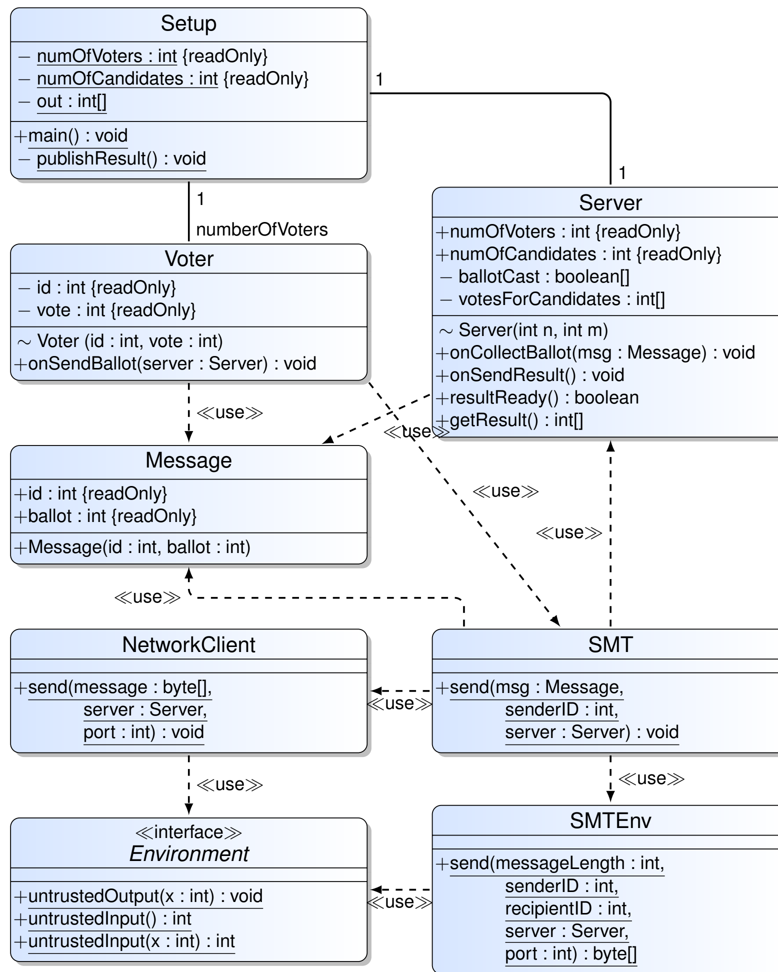

Figure 18.1 UML class diagram of the e-voting system

credentials. We do not need to assume that the network communication is loss-less or must not produce spurious messages.

Listing 18.1 shows the implementation of `Setup#main()`. Essentially, the adversary decides in the loop which client should send its vote next, until the server signals that the result of the election is ready. More precisely, the adversary is modeled through a call to the method `Environment.untrustedInput()`, that decides which client should send its vote. When subsequently the method `onSendBallot()` is called on the corresponding `Voter` object, the client sends its secret vote (stored in the attribute `vote`) to the server (synchronously), with the help of ideal encryption. In its `onCollectBallot()` method, the server immediately counts the vote—provided that the voter did not vote before. Finally, the server is asked by a call to the method

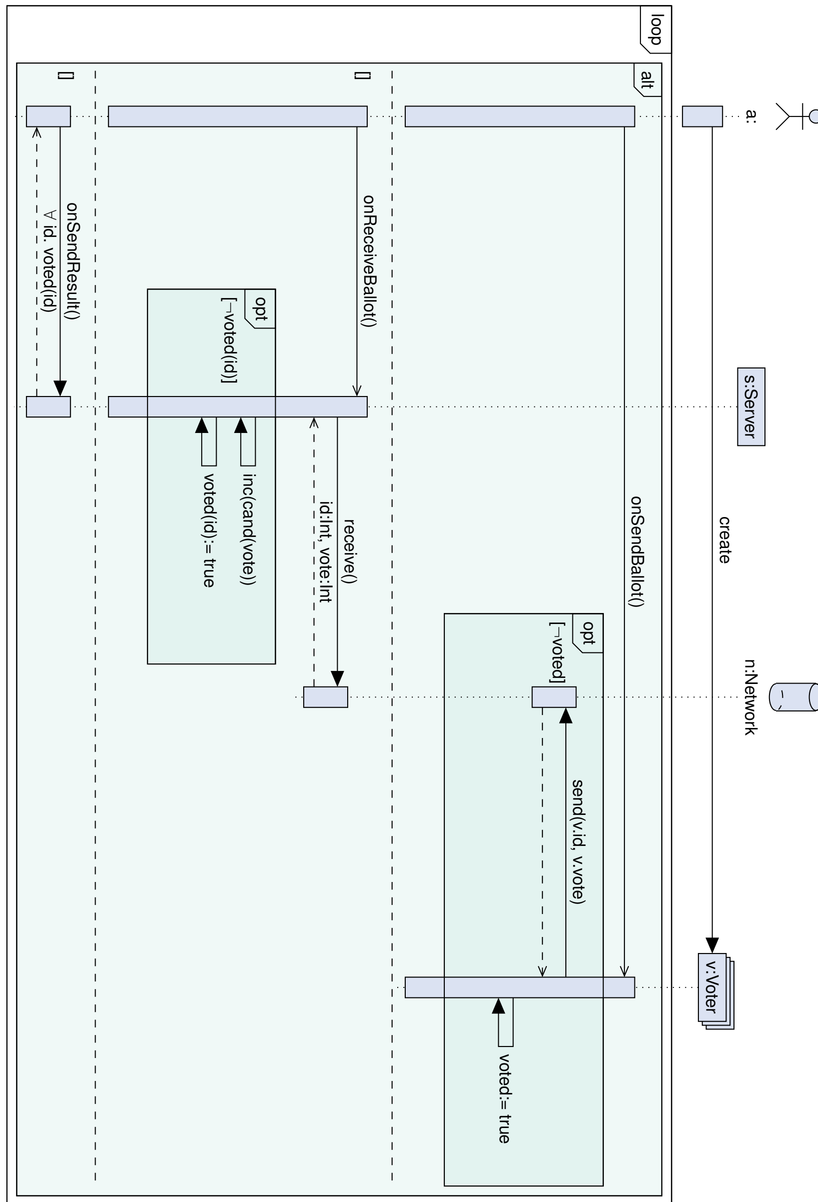

Figure 18.2 The overall protocol of the e-voting system. Here, the actor represent the nondeterministic choice of events.

```
1  /*@ normal_behavior
2    @ requires (\forall int j; 0 <= j && j < numberOfVoters;
3    @                          !server.ballotCast[j]);
4    @ requires (\forall int i; 0 <= i && i < numberOfCandidates;
5    @                          server.votesForCandidates[i]==0);
6    @ ensures (\forall int i; 0 <= i && i < numberOfCandidates;
7    @               server.votesForCandidates[i] ==
8    @                 (\num_of int j; 0 <= j && j < numberOfVoters;
9    @                                voters[j].vote == i));
10   @ diverges true;
11   @*/
12 public void main () {
13     while ( !server.resultReady() ) { // possibly infinite loop
14         // let adversary decide send order
15         final int k = Environment.untrustedInput(voters.length);
16         final Voter v = voters[k];
17         v.onSendBallot(server);
18     }
19     publishResult();
20 }
```

Listing 18.1 Implementation and functional contract for the `Setup#main()` method

`resultReady()` whether the result of the election is ready. If so, the loop terminates and the result is published by a call to the method `publishResult()`.

18.3 Specification and Verification

The overall functional property to prove is that—after all votes have been cast (and collected by the server)—the server posts the correct number of votes per candidate. More precisely, the 'correct number' corresponds to the sum of votes for each candidate as on the ballots filled in by the voters. The information flow property to prove is that no information other than the election result is released.

18.3.1 Specification

The functional contract for the main method is shown in Listing 18.1. In the preconditions, we assume that no voter has cast their vote yet (or more precisely, the server has not yet marked the vote as cast) and all candidates have zero votes (in the server). The postcondition states that the number of votes for each candidate is exactly the number of voters who voted for them. This is expressed using the generalized quantifier `\num_of` (see Section 7.2.2) in Line 8. The explicit `diverges` clause allows this method to not terminate.

The functional specification is augmented with an information flow contract in Listing 18.2. The contract states that—under the condition that the server is initialized correctly—

1. the state of the environment, abstracted by `Environment.envState`, depends at most on its initial value as well as on the number of voters (Line 6);
2. The array `out` itself as well as each of its entries (containing the final result of the election, Lines 8f.) depend at most on
   - the number of candidates,
   - the number of voters, and
   - for each candidate—the correct sum of all votes for them (Lines 10ff.);
3. at most locations of the server, the environment and the result array are changed; and
4. the election might not terminate (because the adversary might block votes of voters forever, Line 16).

The **`\declassifies`** keyword is syntactic sugar (see Section 13.4), but stresses that the array `out` only depends on a well-considered part of the secret—the correct result of the election.

```
1  /*@ normal_behavior
2    @ requires    (\forall int j; 0 <= j && j < numberOfVoters;
3    @                 !server.ballotCast[j]);
4    @ requires    (\forall int i; 0 <= i && i < numberOfCandidates;
5    @                 server.votesForCandidates[i]==0);
6    @ determines  Environment.envState
7    @             \by Environment.envState, numberOfVoters;
8    @ determines  out, (\seq_def int i; 0; out.length; out[i])
9    @             \by numberOfCandidates, numberOfVoters
10   @     \declassifies (\seq_def int i; 0; numberOfCandidates;
11   @                   (\num_of int j;
12   @                       0 <= j && j < numberOfVoters;
13   @                                       voters[j].vote == i));
14   @ assignable  Environment.rep, out,
15   @         server.ballotCast[*], server.votesForCandidates[*];
16   @ diverges true;
17   @*/
18 public void main () { ... }
```

Listing 18.2 Information flow contract for the `main()` method

In order to show that `main()` fulfills its contract, we need a loop invariant (Listing 18.3). In the loop invariant, we talk about a bounded sum (indicated by the keyword **`\num_of`** in Line 4) that is defined through a nontrivial induction scheme: the elements are not added linearly, but only under stuttering and permutation. This makes it—at least the current machinery—impossible to prove the invariant automatically. The information flow part of the loop invariant in Lines 12ff. states that

1. the knowledge of the environment (`Environment.envState`),

```
1  /*@ maintaining \invariant_for(this);
2    @ maintaining (\forall int i; 0 <= i && i < numberOfCandidates;
3    @                 server.votesForCandidates[i] ==
4    @                 (\num_of int j; 0 <= j && j < numberOfVoters;
5    @                           server.ballotCast[j]
6    @                        && voters[j].vote == i));
7    @ maintaining      resultReady
8    @              == (\forall int j; 0 <= j && j < numberOfVoters;
9    @                        server.ballotCast[j]);
10   @ assignable Environment.rep,
11   @          server.ballotCast[*], server.votesForCandidates[*];
12   @ determines Environment.envState, resultReady, numberOfVoters,
13   @            (\seq_def int i; 0; numberOfVoters;
14   @                      server.ballotCast[i])
15   @            \by \itself;
16   @*/
```

Listing 18.3 Loop invariant for the loop in `Setup#main()`

2. the fact whether the result of the election is ready,
3. the number of voters, and
4. the information which voter has already voted (stored in the cells of the array `Server#ballotCast`)

depend at most on

(a) the initial knowledge of the environment,
(b) whether the result of the election initially was ready,
(c) the initial number of voters, and
(d) the initial information which voter has already voted.

As the field `Setup.out` is not modified by the loop, it does not need to be mentioned explicitly in the loop invariant. The fact that the array `Setup.out` itself as well as each of its entries depend at most on

- the number of candidates,
- the number of voters, and
- for each candidate the correct sum of all respective votes for them

can be derived from the contract of `publishResult()` (Listing 18.4) in combination with the assurance of the loop invariant that the server calculates the result correctly. Note that the functional knowledge that the server calculates the result correctly is necessary for proving the declassification. Here, the tight integration of functional and information flow-verification in our approach pays off.

The preservation of the loop invariant is proved with the help of contracts for the methods `untrustedInput()`, `onSendBallot()` and `resultReady()`. The method `untrustedInput()` is declared in the interface `Environment` (Listing 18.5). This interface provides the connection to the environment which is controlled by the attacker. It models all global sources and sinks. The state of the environment is encapsulated in a (ghost) field of type sequence. As any computable

```
/*@ normal_behavior
  @ requires (\forall int i; 0 <= i && i < numberOfCandidates;
  @             server.votesForCandidates[i] ==
  @                (\num_of int j; 0 <= j && j < numberOfVoters;
  @                    voters[j].vote == i));
  @ assignable out;
  @ determines  out, (\seq_def int i; 0; out.length; out[i])
  @        \by  numberOfCandidates, numberOfVoters,
  @             server.votesForCandidates
  @        \declassifies (\seq_def int i; 0; numberOfCandidates;
  @                         (\num_of int j;
  @                               0 <= j && j < numberOfVoters;
  @                               voters[j].vote == i));
  @*/
private void publishResult () { ... }
```

Listing 18.4 Information flow contract of `publishResult`.

information can be encoded into a sequence of integers, this is a valid abstraction. Each method of the `Environment` has a contract which, in essence, guarantees that the environment cannot access any other part of the e-voting system. More precisely, each method is required to meet the following restrictions: 1. The final state of the environment depends at most on its initial state and the parameters of the method. 2. If the method has a result value, then also this result value depends at most on the initial state of the environment and the parameters of the method. 3. At most the state of the environment (represented by field `envState`) is modified. The untrusted input from the environment needs to be sanitized, but still the main loop may not terminate as voters are requested to cast their votes for an arbitrary number of times.

The specification of `Environment` in Listing 18.5 establishes evidence that the information flow specification and verification approach presented in Chapter 13 can be used for the specification and verification of interfaces and consequently also for the specification and verification of open and interactive systems.

The method `Voter#onSendBallot()` generates a new message containing the vote of the voter and sends it over the network as shown in Listing 18.6. The network component is modeled by the classes `NetworkClient` and `SMT` (for ‘secure message transfer’). In the implementation, they mainly encapsulate a single message. `Setup#onSendBallot()` has two contracts. Both require that the invariant of the server holds, and they ensure that the final state of the environment depends at most on its initial value. They differ in the functional part: the first contract additionally requires that the voter has not voted yet. In this case, the contract ensures that the server counted the vote correctly by incrementing the value of `Server#votesForCandidates[vote]`. The second contract requires that the voter did already vote and guarantees in this case that the server does not count the vote again.

The complete specification (for both functional correctness and information flow security) of the system consists of approximately 270 lines of JML.

```
public interface Environment {
    //@ public static ghost \seq envState;

    //@ public static model \locset rep;
    //@ public static represents rep = \locset(envState);
    //@ accessible rep : \locset(envState);

    /*@ normal_behavior
      @ ensures     true;
      @ assignable  rep;
      @ determines  Environment.envState, \result
      @             \by Environment.envState;
      @*/
    //@ helper
    public static int untrustedInput();

    /*@ normal_behavior
      @ ensures     true;
      @ assignable  rep;
      @ determines  Environment.envState
      @             \by Environment.envState, x;
      @*/
    //@ helper
    public static void untrustedOutput(int x);

    /*@ normal_behavior
      @ ensures     0 <= \result && \result < x;
      @ assignable  rep;
      @ determines  Environment.envState, \result
      @             \by Environment.envState, x;
      @*/
    //@ helper
    public static int untrustedInput(int x);

}
```

Listing 18.5 Declaration of the interface `Environment`.

18.3.2 Verification

For the functional verification of this implementation, there are 13 methods to be considered, with a total of 150 lines of (executable) code and approximately 140 lines of specification. The specification includes class invariants, method contracts, and loop invariants. Given our overall experience in formal specification, a 1:1 ratio of code against specification seems reasonable. Most method contracts can be proven without much effort. For instance, the proof of the `Voter#onSendBallot()` method

```
/*@ normal_behavior
  @ requires    ! server.ballotCast[id];
  @ requires    \invariant_for(server);
  @ ensures         server.votesForCandidates[vote]
  @              == \old(server.votesForCandidates[vote])+1;
  @ ensures     server.ballotCast[id];
  @ assignable  server.votesForCandidates[vote],
  @             server.ballotCast[id], Environment.rep;
  @ determines  Environment.envState \by \itself;
  @ also normal_behavior
  @ requires    server.ballotCast[id];
  @ requires    \invariant_for(server);
  @ ensures             \old(server.votesForCandidates[vote])
  @                  == server.votesForCandidates[vote];
  @ ensures     \old(server.ballotCast[id])
  @                  == server.ballotCast[id];
  @ assignable  Environment.rep;
  @ determines  Environment.envState \by \itself;
  @*/
public void onSendBallot(Server server) {
    Message message = new Message(id, vote);
    //@ set message.source = this;
    SMT.send(message, id, server);
}
```

Listing 18.6 Contract of `Voter#onSendBallot()`

consists of 2,400 proof steps and takes 6 s, performed by the KeY prover without further interaction.[3]

Mostly due to its unconventional loop condition, the `main()` method could not be verified automatically. To prove equality of sums, we had to apply the split_sum rule several times interactively. This rule rewrites a sum comprehension into two comprehensions over split ranges. In addition, we have added some rules representing lemmas dealing with bounded sums to the rule base of KeY; and we have proven their soundness. The proof for `main()` finally took about 63,000 proof steps, only ten of which were applied by hand. The computation time for the automated parts of the proofs was 580 s.[4]

[3] Time measurements have been taken on standard desktop computer (1 processor core, 1.5 GHz, 4 GiB RAM, Debian/Linux).

[4] Please note that it is difficult to give figures for manual proofs. Firstly, the human interaction is necessary and therefore cannot be compared against computation time. Secondly, the time for the remaining automated rule application is not reliable as it may include time for rules applied automatically, but reverted by the user.

Information Flow Analysis

The subsequent verification of the information flow properties of the system took about four days. The final information flow proof consists of 23 subproofs with about 7,800 proof steps including some user interactions. The optimizations described in Section 13.5.1 have proven to be indispensable for the scalability of the self-composition approach.

18.4 Discussion

In the course of this chapter, we presented an approach to verify a Java implementation of an electronic voting system. Analyses of such systems mostly target the design or the system level. Even a system like the one presented here—which can be considered small if measured in lines of code—poses a major challenge to formal verification at code level. Therefore, it is not surprising that the proofs were laborious.

Actually, far more effort than in conducting the interactive proofs needed to be put into understanding the system and developing an appropriate specification. Apart from representing the high-level design, an appropriate specification needs to be correct w.r.t. the program. This in turn requires early proof attempts with prototype implementations. Our approach to first verify a very basic version and to refine it later on turned out to be helpful in this regard. It provided clear, reachable milestones.

An interesting point is that the main complexity resides in the synthetic setup that is used to model a deployed system and not in the components that are actually used. It is well-known that tools intended for code verification do not perform well at system level verification. As already noted by Woodcock et al. [2008], verifying software that was not originally produced for the purpose of verification almost always constitutes an ill-fated endeavor. While not of the size of system described by Woodcock et al., we experienced this phenomenon in the (original) sElect system by Küsters et al. The starting point of our verification was a final piece of software. In particular, specifications had to be conceived by ourselves, using only the present source code and informal descriptions of the components' behavior. Although there are no guidelines to produce well-verifiable programs, we believe that adherence to common software engineering guidelines would render formal specification and verification more feasible.

18.4.1 A Hybrid Approach to Information Flow Analysis

In order to perform an information flow analysis on a 'more realistic' implementation of the sElect system, Küsters et al. [2015] describe a hybrid approach that combines functional verification in KeY with a lightweight information flow analysis based on *program dependency graphs* [Hammer, 2009]. In order to get the JOANA tool [Graf

et al., 2013] to accept declassification, the original program is transformed such that it does not have any illegal information flow by construction.

This technique is based on a simulation of noninterference in the Java code. The secret here is only a single bit (stored in the static field `Setup.secret`). In the setup, two arrays of voter objects are created according to the environment to simulate two possible high inputs. The program aborts in case they yield nonequivalent results. At this point in the program execution, both high inputs are incomparable modulo the declassified property (i.e., the result of the election). Then one array is chosen, depending on the secret, to be used in the main loop.

Since the functional property and the actual implementation did not change in comparison to Section 18.3.2, there are only new verification targets, namely 1. the `Setup()` constructor, that establishes the above described setup and 2. the so-called 'conservative extension' method, that is called after the election has terminated. The extension effectively eliminates the declassification through overwriting the result, as computed by the actual implementation, with a precomputed correct result. The central goal was to prove that this extension is really ineffective (which is an even stronger property than conservatism).

Both require significant interaction in proving, while having the automated prover apply several thousands of rules in between each interactive step. Interestingly, this is mainly due to the sheer size of the code under investigation, but not to any particularly pattern that is hard to prove. After all, the proof for `main()` consists of over 200,000 proof steps, of which some 100 were applied by hand. The labor invested in verifying it approximately amounts to three weeks full time.

18.4.2 Related Work

To the best of our knowledge, this is the first time that preservation of privacy of votes could be shown *on the code level* for a (simple) e-voting system. Systems like Bingo Voting [Bohli et al., 2009] Civitas [Clarkson et al., 2008], Helios [Adida, 2008], or Scantegrity [Chaum et al., 2009]—which are much more elaborate—provide guarantees on the design level, but it is not clear whether their implementations preserve these guarantees. Clarkson et al. [2008] mention that their Civitas system has been checked for information flows with JIF [Myers, 1999], but it is not stated clearly which properties have been checked.

Bär [2008] specified functional properties of a Java implementation of the Bingo Voting system with the Java Modeling Language. These specifications have been partially checked with the (unsound and incomplete) ESC/Java2 tool by Beck [2010]. Kiniry et al. [2006] report on the Dutch KOA remote voting system, that has been used in the European Parliament election in 2004 for a small group of voters. In order to specify the (offline) vote counting module with JML and subsequently analyze it with ESC/Java2, they reimplemented the KOA system in Java.

While using ideal cryptographic functionality in code verification can be seen as state of the art, there are other approaches that include formal reasoning about cryp-

tographic guarantees [Stern, 2003]. Barthe, Grégoire, and Béguelin [2009] present a framework in which adversaries can be modeled as probabilistic polynomially bounded `while` programs. A probabilistic relational Hoare logic—extending Benton's logic [2004]—allows one to formally reason about these adversaries, that is implemented in the EasyCrypt system [Barthe et al., 2013b].

18.4.3 Conclusion

This case study clarified the boundaries to which verification scales with the KeY prover. Going even further, we performed first experiments with replacing synchronous by asynchronous message transfer. Again, the client and server components can be verified with reasonable effort, but the setup is largely intractable.

Nevertheless, this case study serves as a benchmark and has pushed forward several performance improvements in the KeY system. This includes both improvements in the strategy (i.e., moving to a more tractable complexity class) and practical implementation changes.

The e-voting case study shows that precise information flow verification techniques as the ones presented in Chapter 13 are essential for the verification of complex information flow properties, in particular for the verification of semantic declassification. It also shows that the optimizations introduced in Section 13.5.1 are indispensable for the feasibility of the self-composition approach.