Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2014,11
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

Formal Verification of an Electronic Voting System

Daniel Bruns

2014

KIT – University of the State of Baden-Wuerttemberg and National Research Center of the Helmholtz Association

Fakultät für **Informatik**

Please note:
This Report has been published on the Internet under the following Creative Commons License:
http://creativecommons.org/licenses/by-nc-nd/3.0/de.

Formal Verification of an Electronic Voting System*

Daniel Bruns

August 4, 2014

Abstract

Electronic voting (e-voting) systems that are used in public elections need to fulfil a broad range of strong requirements concerning both safety and security. Among these requirements are reliability, robustness, privacy of votes, coercion resistance and universal verifiability. Bugs in or manipulations of an e-voting system may have considerable influence on the life of the humans living in a country where such a system is used. Hence, e-voting systems are an obvious target for software verification.

In this paper, we report on an implementation of such a system in Java and the formal verification of functional properties thereof in the KeY verification system. 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 votes, the server calculates the correct votes for each candidate w.r.t. the original ballots. This kind of trace property is difficult to prove with static techniques like verification and typically yields a large specification overhead.

Contents

1 Electronic Voting **4**

2 Setup **4**
- 2.1 Verification Approach 4
- 2.2 System Overview 5
- 2.3 Verification of a Nonmodular Software System 6

3 Implementations and Verification **8**
- 3.1 Basic System 9
- 3.2 Adding a Network Component 9
- 3.3 Hybrid Approach Setup 11

4 Conclusion **12**

*This work has been supported by Deutsche Forschungsgemeinschaft (DFG) under project "Program-level Specification and Deductive Verification of Security Properties (DeduSec)" within SPP 1496 "Reliably Secure Software Systems (RS^3)".

1 Electronic Voting

Elections form a part of everyday life that has not (yet) been conquered by computerized systems. This is partly due to the relatively high effort—elections do not occur often—and partly to little public trust in 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 those, 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 on paper ballots. This includes *confidentiality* of individual votes. In particular they must not be attributed to a 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. In electronic voting, the idea is to issue a receipt to the voter, a so-called *audit trail*, for casting their vote. After the votes have been tallied, they 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 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 bribe). The idea is that trail and bulletin board are of a shape such that an attacker cannot distinguish the vote even under the circumstance that the coerced voter is trying to reveal it. This way, electronic voting may be even more secure than voting using paper ballots.[1]

Due to this nature of requiring highest security guarantees, electronic voting has been frequently named a natural target for verification, e.g., by Clarkson et al. [2008], Cortier [2014].

2 Setup

We consider parts of the electronic voting system described by Küsters et al. [2011]. In this system, remote voters can cast one vote for some candidate. This vote is sent through a secure channel to a tallying server. The secure channel functionality 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 votes for each candidate—once all voters have cast their vote.

2.1 Verification Approach

As described in [Beckert et al., 2012], the distant goal is to show that no confidential information (i.e., votes) are leaked to the public. Obviously, the result—

[1] An important practical aspect of elections is *fairness*. As argued in [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.

a public information—does depend on confidential information. And 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*. Beckert et al. describe how such a property can be formalized using Java Dynamic Logic and proven in the KeY verification system [Beckert et al., 2007, Ahrendt et al., 2014].

Declassification—in the sense that parts of the secret is information is purposely released[2]—is essentially a functional property. In general, it cannot be dealt with using lightweight static analyses, such as type systems or program dependency graphs, that are still predominant in the information flow analysis world. Instead, it requires semantically precise information flow analyses as provided by the direct formalization of noninterference in dynamic logic [Amtoft and Banerjee, 2004, Darvas et al., 2005, Scheben and Schmitt, 2012]. In fact, this functional verification can be decoupled from the verification of information flow properties. Here, we report on functional verification only.

There are two approaches to verify information flow properties in this system. The first one, described in [Scheben, 2014, Chap. 9], is based on dynamic logic formalization of noninterference and theorem proving in KeY as layed out by Scheben and Schmitt. Scheben claims that this is the "first time that preservation of privacy of votes could be shown *on the code level*." The proof still relies on functional correctness established by the proofs described in Sect. 3.2 of this paper.

Another approach combines functional correctness proofs in KeY with lightweight static information flow analysis [Küsters et al., 2013]. The target program is transformed in such a way that there is no declassification of information. We then prove that this transformation preserves the original functional behavior. This is discussed in Sect. 3.3. It allows the static analyzer JOANA [Hammer, 2009, Graf et al., 2013]—which is sound, but incomplete—to report absence of information flow.

The system uses cryptography and other security mechanisms. From a functional point, 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 polynominally bounded[3]—this is called a Dolev/Yao attacker [Dolev and Yao, 1983]—can not be reasonably formalized. As a fact, even encrypted transmission does leak information. We use the common technique to replace the actual encryption by *ideal* encryption, that just produces constant values and stores the secret message in a way that it can be retrieved in ideal decryption. For more detail, the interested reader is kindly pointed to [Küsters et al., 2011].

2.2 System Overview

The basic protocol works as follows: First, voters register their clients (represented by a class `Voter` here), 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 voter identifier and vote) or

[2] This understanding is opposed to other uses of the term 'declassification' that denote the release of *any* information under certain constraints.

[3] As an aside, secrecy against this class of attacker can only be given under the assumption that the $\mathcal{P} \neq \mathcal{NP}$ conjecture [Cook, 1971] is valid.

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 this 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[4], and only then a result (i.e., the number of votes per candidate) is presented. This is illustrated in the sequence diagram in Fig. 1; the actor here represent the indeterministic choice of events.

This simplified representation hides many aspects essential to real systems. We assume both a working identification and that identities cannot be forged. And we assume that the network does not leak any information on the ballot (i.e., voter identifier and vote). This is meant to be assured through means of cryptography. It may leak—and probably will in practice—other information such as networking credentials. We do not need to assume that the network is lossless or must not produce spurious messages.

2.3 Verification of a Nonmodular Software System

The particular challenge in this case study is that we prove a highly nonlocal (both spatial and temporal) property: After the election is closed, the *original* votes of all voters who are marked as voted *in the server* are counted to the result. This property is spatially nonlocal since it refers to the server and all voters simultaneously. It is temporally nonlocal since it refers to a particular state. This is very much countering the idea of Design by Contract [Meyer, 1992], where properties are local to method call (and return) events. Instead, we have a kind of a trace property, that needs to be proven for every run of the protocol.

To verify this in the implementation, runs of the protocol are simulated through Java code again. Then we can annotate the synthetic main method with the desired property. As we will see, simulation in Java brings with it the whole 'clutter' of a real-world language, such as object identities, createdness, heap separation, etc. Many of the specification items intended for the main method need to 'tracked' through the program stack trace. This approach comes with some major disadvantages. Firstly, the resulting specifications are strongly specialized and probably cannot be reused. Secondly, it produce a high specification overhead and thus also a verification overhead. Finally, reasoning about Java programs is far more expensive than reasoning on an abstract level. For KeY, though performing symbolic execution is not a bottleneck, reasoning about heap allocated data definitely is.

Example 1. *Consider the following problem: The entries of two integer vectors (of fixed length) are nonnegative, prove that the vector resulting from pairwise addition again contains only nonnegative entries. This is more or less obvious; and a formalization in first order logic can be proven in KeY in less than 100 rule applications, taking* 0.1 s *time on a standard desktop computer. Now we implement this in Java, using arrays as vector representation, as shown in Listing 1.*[5] *The first thing to notice is the outright specification overhead, including a*

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

[5] We added (weak) purity and freshness of the result to the postcondition to make the method 'more functional.'

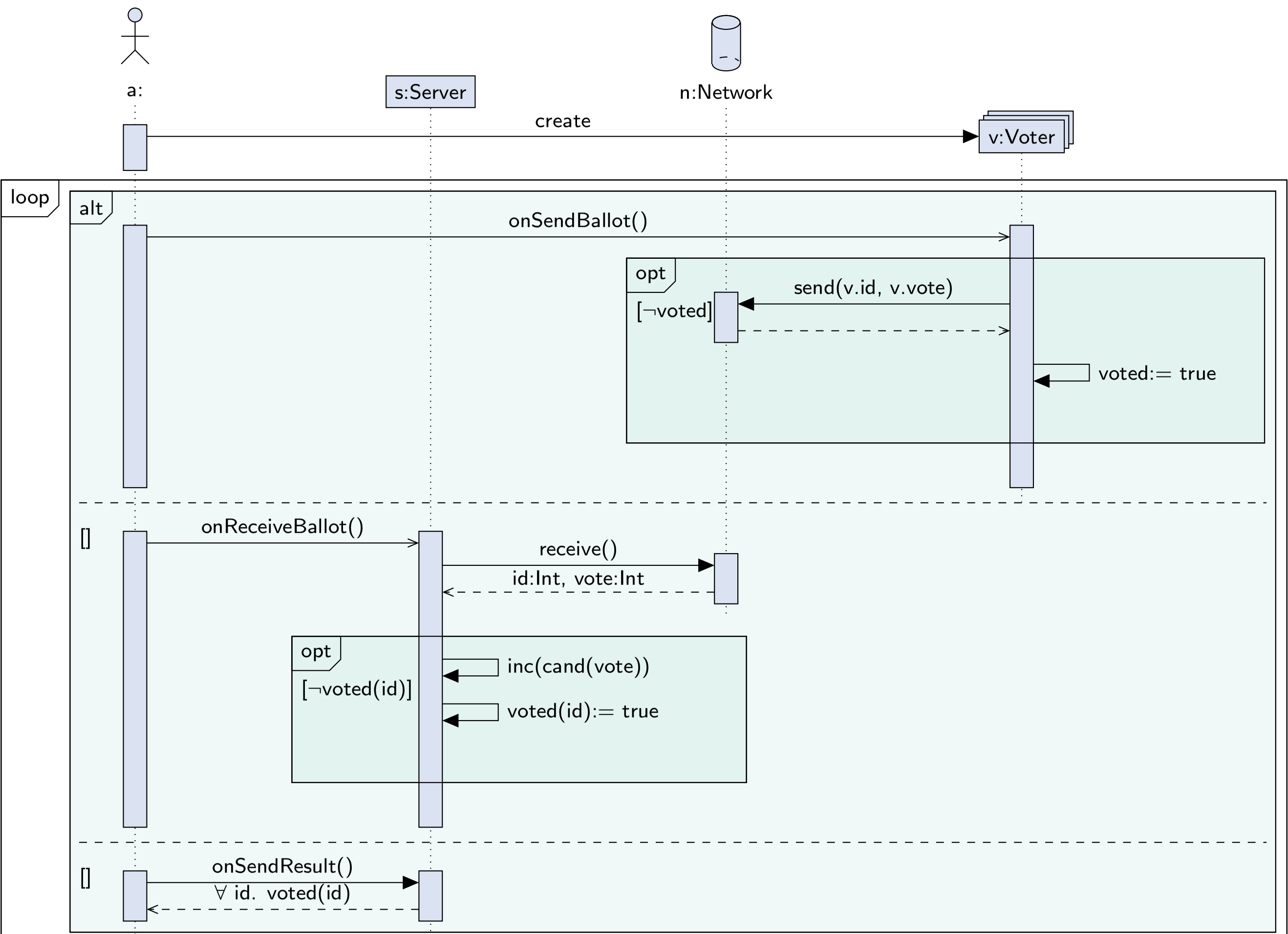

Figure 1: The overall protocol of the e-voting system

```
class VectorAdd {
    int[] a, b;

    /*@ requires a.length == b.length;
      @ requires (\forall int j; 0 <= j && j < a.length;
      @                a[j] >= 0 && b[j] >= 0);
      @ ensures (\forall int j; 0 <= j && j < a.length;
      @               \result[j] >= 0);
      @ ensures \fresh(\result);
      @ pure
      @*/
   int[] add() {
       int[] c = new int[a.length];
       /*@ maintaining 0 <= i && i <= a.length;
         @ maintaining (\forall int j; 0 <= j && j < i;
         @                  c[j] == a[j] + b[j]);
         @ maintaining \fresh(c);
         @ assignable c[*];
         @ decreasing a.length-i;
         @*/
       for (int i=0; i<a.length; i++)
           c[i] = a[i]+b[i];
       return c;
   }
}
```

Listing 1: A simple Java program implementing vector addition

loop invariant and frame annotations. The shown method can be proven correct w.r.t. the given specification automatically in KeY. But the proof size is considerably larger than for the FOL version. It now takes over 6500 rule applications and 11.5 s *time.*

Our approach to tackle the complexity of the system is to verify a heavily reduced version first, then to refine it stepwise. This way, only smaller components are changed and the modular verification paradigm enshrined in the KeY system allows us to reuse many of the already obtained proofs while we only verify the changed components. The versions of the system produced in this way were developed by ourselves, in contrast to the actual system implemented by Küsters et al.

3 Implementations and Verification

As described above, the goal is to verify a simple implementation of a distributed e-voting system. The design is based on the system developed by Küsters et al., but reduced to its essential functionality. We start with a very basic version and incrementally add functionality or modeling aspects. Each step includes formal specification in the Java Modeling Language (JML) [Leavens et al., 2006] and a full functional verification using a development version of KeY (pre-2.2).

For some of the proofs, we have given figures on the number of proof steps and the computation time for automated rule application. Automated rule ap-

plication in KeY is supposed to be deterministic. Therefore, given the same version of KeY and the same options, the figure for proof steps should be verifiable in new experiments. Time measurements have been taken on a standard laptop computer (1 processor core, 1.5 GHz, 4 GB RAM, Linux). Another version of KeY, in particular the 2.2 release, may yield other figures. 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.

In any implementation, there is a class `Setup` that contains the main loop, that contains all global actions. 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, i.e., the sum of votes for each candidate on the original ballots held by voters.

3.1 Basic System

In the basic implementation, there are classes `Server` and `Voter` (clients) as well as a `Message` encapsulation class. Voters cast their votes in the order in which they are defined (and exactly once). Messages are passed directly to the server (through a method call).

The main method along with JML specifications is shown in Listing 2. 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`. The explicit `diverges` clause allows this method to not terminate. The present implementation actually do terminate, but since the implementations shown below do not terminate, we only require partial correctness for the sake of consistent properties.

Since the loop is based on simple linear traversal of an array, the invariant is essentially an abstraction from the contract. The server entries for ballots cast and votes for candidates are the only changed locations here.

In this version, there are 4 methods to be verified with a total of 18 lines of (executable) code and approximately 80 lines of specification.[6] The specification includes class invariants, method contracts, and loop invariants. Given our overall experience in formal specification, a 1:4 ratio of code against specification seems reasonable. The implementation can be verified fully automatically, but the proof for the main method contains over 27,000 proof steps and took 210 s of computation time.

3.2 Adding a Network Component

To model a more realistic system, in the second implementation, we allow the adversary to decide on the order of events (i.e., voter submits a ballot, server collects a ballot, election ends). We now have an explicit modeling of a network component, through which messages are sent. However, the implementation is based on synchronous communication as the server immediately fetches a message that has been sent. This is the version on which Scheben [2014] reports.

[6]Since there is no canonical representation, specification cannot be quantified objectively.

```
/*@ normal_behavior
  @ requires (\forall int j;
  @              0 <= j && j < numberOfVoters;
  @              !server.ballotCast[j]);
  @ requires (\forall int i;
  @              0 <= i && i < numberOfCandidates;
  @              server.votesForCandidates[i]==0);
  @ ensures (\forall int i;
  @              0 <= i && i < numberOfCandidates;
  @              server.votesForCandidates[i] ==
  @                  (\num_of int j;
  @                      0 <= j && j < voters.length;
  @                      \old(voters[j].vote) == i));
  @ diverges true;
  @*/
public void main () {
    /*@ maintaining \invariant_for(this);
      @ maintaining 0 <= k && k <= numberOfVoters;
      @ maintaining (\forall int j;
      @                  0 <= j && j < numberOfVoters;
      @                  j < k <==> server.ballotCast[j]);
      @ maintaining (\forall int i;
      @                  0 <= i && i < numberOfCandidates;
      @                  server.votesForCandidates[i] ==
      @                  (\num_of int j;
      @                      0 <= j && j < k;
      @                      voters[j].vote == i));
      @ assignable server.ballotCast[*],
      @            server.votesForCandidates[*];
      @*/
    for (int k= 0; k < voters.length; k++) {
        server.onCollectBallot(voters[k].onSendBallot());
    }
    server.onSendResult();
}
```

Listing 2: The main loop in the basic setup

The main loop is changed such that the order in that voters cast their votes is decided by the environment (low input). We have an additional class `Environment` that models all global sources and sinks. The untrusted input from the environment needs to be sanitized, but still the main loop may not terminate and voters are requested to cast their votes for an arbitrary number of times. The classes `NetworkClient` and `SMT` (for ‘secure message transfer’) model the network component. In the implementation, they mainly encapsulate a single message. Except for these additional classes, the size of the system is about the same as above in Sect. 3.1.

For the specification effort, this means that we need refined contracts that take into account the situation that voters have already cast votes. In the loop invariant—which is still the one displayed in Listing 2—we talk about a bounded sum which is defined through a nontrivial induction scheme. The elements are not added linearly, but only under stuttering and permutation. This makes it (at the moment) impossible to prove the invariant automatically. To prove equality of sums, we had to apply the ‘split sum’ rule several times interactively. In addition, we have added some lemma rules dealing with bounded sums to the rule base of KeY and we have proven them sound. The proof for `main` then took about 63,000 proof steps, of which less than 10 were applied by hand. Computation time for the automated parts of the proofs was 580 s.

The specification of the `Voter#onSendBallot()` method has changed in comparison to Sect. 3.1. Its proof is slightly larger—from 600 proof steps and 1.3 s in the basic version to 2400 proof steps and 6 s—but the KeY prover was still able to find the proof automatically. All other methods were not touched; and their respective proof is still valid.

3.3 Hybrid Approach Setup

Küsters et al. [2013] describe a so-called “hybrid approach” that combines functional verification in KeY with lightweight, flow-insensitive, information flow analysis in JOANA. In order to leverage JOANA to accept declassification, the original program is transformed such that it does not have any flow.

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. This setup can be seen in Listing 3.

Since the functional property and the actual implementation has not changed in comparison to Sect. 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 shown in Listing 4, that is called after the election has terminated. It effectively eliminates the declassification through overwriting the result, as computed by the actual implementation, with a precomputed correct result.

Both required significant interaction in proving, while having the automated prover apply some thousands of rules in between each interactive step. Interestingly, this is mainly due to the sheer size of the code under investigation.

```
1       private Setup () {
2           final int n = numberOfVoters;
3           final int m = numberOfCandidates;
4
5           // let adversary create fake voters
6           Voter[] v1 = createFakeVoters();
7           Voter[] v2 = createFakeVoters();
8           int[] r1 = computeResult(v1);
9           int[] r2 = computeResult(v2);
10          if (equalResult(r1,r2)) {
11              // store correct result
12              out = r1;
13
14              // select voters according to secret
15              voters = secret? v1: v2;
16
17              server = new Server(n, m);
18          } else
19              // abort if not equal
20              throw new Throwable();
21      }
```

Listing 3: The “hybrid approach” setup

By ‘size,’ we do not only understand single lines of code, as often in software analysis, but rather the lack of proper modularization. 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.

4 Conclusion

We have presented an approach to verify a Java implementation of an electronic voting system. As Scheben [2014] states, analyses of such systems mostly target the design or the system level. Even a system like the one presented here—

```
/*@ requires out == computeResult(voters);
  @ requires
  @     (\forall int i; 0 <= i && i < numberOfCandidates;
  @         server.votesForCandidates[i] ==
  @         (\num_of int j; 0 <= j && j < numberOfVoters;
  @             voters[j].vote == i));
  @ ensures equalResult(out,\old(out));
  @*/
private void conservativeExtension () {
    out = server.votesForCandidates;
}
```

Listing 4: “Conservative extension” in the “hybrid approach” setup

which can be considered small, in particular if measured in lines of code—poses a major challenge to formal verification at code level. It is not surprising that the proofs were laborious.

Actually, far more effort than in conducting the interactive proofs has been 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 proof attempts. Our approach to first verify a very basic version and to refine it later has been proven to be helpful in this regard. It provided clear milestones, that were actually reachable.

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 constitutes an almost doomed endeavor. While not the size of system described by Woodcock et al., we have experienced this phenomenon here. The starting point for verification was a final piece of software, without any formal development process behind it. In particular, specifications had to be conceived by ourselves, using only the present source code and informal descriptions of the components' behavior. Although there no guidelines to produce well verifiable programs, we believe that adherence to common software engineering guidelines would render formal specification and verification more feasible.

This case study has made clear the boundaries to which verification scales with the KeY prover. Going even further, we have made 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. In particular, some proofs forced KeY to consume a lot of memory. In the past, memory has never been the limiting force in proofs, but here KeY used up to 40 GB of RAM. Later improvements in the implementation found by the author reduced memory consumption by 30–40% on proofs of this size. These improvements have played a large part in the development of the milestone release KeY 2.2 in April 2014.

References

Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt, and Mattias Ulbrich. The KeY platform for verification and analysis of Java programs. In Dimitra Giannakopoulou and Daniel Kroening, editors, *Verified Software: Theories, Tools, and Experiments (VSTTE 2014)*, Lecture Notes in Computer Science. Springer-Verlag, 2014. To appear.

Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In Roberto Giacobazzi, editor, *11th Static Analysis Symposium (SAS), Verona, Italy*, volume 3148 of *LNCS*, pages 100–115. Springer-Verlag, 2004.

Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 2007.

Bernhard Beckert, Daniel Bruns, Ralf Küsters, Christoph Scheben, Peter H. Schmitt, and Tomasz Truderung. The KeY approach for the cryptographic verification of Java programs: A case study. Technical Report 2012-8, Department of Informatics, Karlsruhe Institute of Technology, 2012. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000027497.

Daniel Bruns. Elektronische Wahlen: Theoretisch möglich, praktisch undemokratisch. *FIfF-Kommunikation*, 25(3):33–35, September 2008. ISSN 0938-3476.

Michael R. Clarkson, Stephen Chong, and Andrew C. Myers. Civitas: Toward a secure voting system. In *IEEE Symposium on Security and Privacy*, pages 354–368. IEEE Computer Society, 2008. URL http://doi.ieeecomputersociety.org/10.1109/SP.2008.32.

Stephen A. Cook. The complexity of theorem-proving procedures. In *Conference record of third annual ACM symposium on theory of Computing*, pages 151–158, Shaker Heights, Oh., 1971. ACM.

Véronique Cortier. Electronic voting: how logic can help. In Stéphane Demri, Deepak Kapur, and Christoph Weidenbach, editors, *Int. Joint Conference on Automated Reasoning 2014*, volume tba of *LNCS*. Springer, 2014.

Ádám Darvas, Reiner Hähnle, and Dave Sands. A theorem proving approach to analysis of secure information flow. In Dieter Hutter and Markus Ullmann, editors, *Proc. 2nd International Conference on Security in Pervasive Computing*, volume 3450 of *Lecture Notes in Computer Science*, pages 193–209. Springer-Verlag, 2005.

Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

Jürgen Graf, Martin Hecker, and Martin Mohr. Using JOANA for information flow control in Java programs – A practical guide. In Stefan Wagner and Horst Lichter, editors, *Software Engineering (Workshops)*, volume 215 of *Lecture*

Notes in Informatics, pages 123–138. Gesellschaft für Informatik, 2013. ISBN 978-3-88579-609-1.

Christian Hammer. *Information Flow Control for Java – A Comprehensive Approach based on Path Conditions in Dependence Graphs*. PhD thesis, Universität Karlsruhe (TH), July 2009. URL http://digbib.ubka.uni-karlsruhe.de/volltexte/1000012049.

Ralf Küsters, Tomasz Truderung, and Andreas Vogt. Verifiability, Privacy, and Coercion-Resistance: New Insights from a Case Study. In *Proceedings of the 32nd IEEE Symposium on Security and Privacy (S&P)*, pages 538–553, Oakland, California, USA, 2011. IEEE Computer Society.

Ralf Küsters, Tomasz Truderung, Bernhard Beckert, Daniel Bruns, Jürgen Graf, and Christoph Scheben. A hybrid approach for proving noninterference and applications to the cryptographic verification of Java programs. In Christian Hammer and Sjouke Mauw, editors, *Grande Region Security and Reliability Day 2013*, Luxembourg, 2013. URL http://grsrd.uni.lu/papers/grsrd2013_submission_2.pdf. Extended Abstract.

Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006. URL http://doi.acm.org/10.1145/1127878.1127884.

Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, October 1992.

Christoph Scheben. *Program-level Specification and Deductive Verification of Security Properties*. PhD thesis, Karlsruhe Institute of Technology, 2014. in preparation.

Christoph Scheben and Peter H. Schmitt. Verification of Information Flow Properties of Java Programs without Approximations. In Bernhard Beckert, Ferruccio Damiani, and Dilian Gurov, editors, *Formal Verification of Object-Oriented Software*, volume 7421 of *Lecture Notes in Computer Science*, pages 232–249. Springer, 2012.

Jim Woodcock, Susan Stepney, David Cooper, John A. Clark, and Jeremy Jacob. The certification of the mondex electronic purse to ITSEC level E6. *Formal Asp. Comput*, 20(1):5–19, 2008. URL http://dx.doi.org/10.1007/s00165-007-0060-5.