Poster: Security in E-Voting

Daniel Bruns†, Huy Quoc Do‡, Simon Greiner†, Mihai Herda†, Martin Mohr†, Enrico Scapin*, Tomasz Truderung*, Bernhard Beckert†, Ralf Küsters*, Heiko Mantel‡ and Richard Gay‡
*University of Trier, lastname@uni-trier.de, †Karlsruhe Institute of Technology, firstname.lastname@kit.edu
‡Technische Universität Darmstadt, do@rbg.informatik.tu-darmstadt.de, {gay,mantel}@mais.informatik.tu-darmstadt.de

I. Introduction

In this work, we consider the verification of security properties of Java programs that use cryptographic operations. Our motivating objective is to provide cryptographic guarantees on the code level for a Java implementation of a realistic, usable electronic-voting system, called *sElect*[1].

While tools for verification of Java programs are available, including tools for checking noninterference properties, they cannot deal with cryptography: guarantees for cryptographic primitives are based complexity arguments and therefore these primitives do not provide absolute security against unbounded adversaries (which is the adversary model employed in these tools).

In our work, we propose a framework that allows the existing tools for checking noninterference to verify cryptographic properties, such as cryptographic indistinguishability, of Java programs. We applied our framework to several case studies, using the Joana and KeY tools. The integration of these tools in our framework led to interesting insights and motivated us to improve the existing techniques and develop new techniques for proving noninterference.

II. Approach

A. The CVJ framework

We have developed a framework for the cryptographic analysis of Java programs (the CVJ framework) [1]. This framework enables existing tools that can check (standard) noninterference properties for Java programs, but a priori cannot deal with cryptography, to establish cryptographic indistinguishability properties for Java programs. The CVJ framework combines techniques from program analysis and universal composability [2], [3], a well-established concept in cryptography. The idea is to first check noninterference properties for the Java program to be analyzed where cryptographic operations (such as encryption) are performed within so-called *ideal functionalities*. Such functionalities typically provide guarantees even in the face of unbounded adversaries and can often be formulated without probabilistic operations and, therefore, they can be carried out by tools that a priori cannot deal with cryptography (probabilities, polynomially bounded adversaries). The results of the framework now imply that the original Java program (using actual cryptographic operations) enjoys strong cryptographic indistinguishability.

As to checking non-interference, many program analysis tools can only deal with closed Java programs. The systems to be analyzed are, however, often open: they interact, for example, with an untrusted (and unspecified) network. Therefore, as part of the framework, we have proposed a proof techniques that enable program analysis tools to verify noninterference properties of open systems.

B. Tools

In our work we use two tools for Java programs, Joana and KeY, in combination with the CVJ framework. In this section we shortly introduce these tools and describe the improvements which are part of this work. In the next section we describe how these tools can be combined in order to prove challenging properties of realistic programs.

Joana[2] [4] is a tool for the fully automatic analysis of noninterference properties of Java programs. It computes a conservative approximation of the information flows inside a given program in form of a *program dependence graph* (PDG). Then, the PDG is checked for illegal information flows using advanced dataflow analysis based on *slicing*. If no illegal flow is found in the PDG, the program is guaranteed to be noninterferent. The correctness of this implication has been verified with a machine-checked proof [5].

Since Joana is an automatic tool which uses overapproximation, it may report information flows which are not actually there (*false alarms*). Joana employs sophisticated program analysis techniques that help to reduce such false alarms. Within this work, we improved these techniques. For instance, we added support for the recognition of *killing definitions*. Recognizing killing definitions can improve precision in situations where a variable which contains a secret value is overwritten before being propagated to a public channel.

KeY[3] [6] is an program verification system, which targets sequential Java. At its core lies an interactive theorem prover for first-order dynamic logic (JavaDL) [7]. Program specifications can be given in the Java Modeling Language (JML) [8]. The sequent calculus for JavaDL that is built into KeY precisely reflects the semantics of sequential Java, i.e., it does not use approximations. Thus, analysis techniques built on KeY are precise. They do not report any false positives. Proofs can be automated to a certain degree, while the user can interact with the prover at any time. KeY can generate counter examples and unit tests from failed proof attempts.

[1] *sElect*, secure Election: https://github.com/ttruderung/sElect

[2] The sourcecode of Joana is available at `http://joana.ipd.kit.edu/`.

[3] KeY is free software and can be downloaded from http://key-project.org/.

C. The Hybrid approach.

Fully automated tools are often preferable over interactive since with such tools program analysis is typically less time-consuming and requires less expertise. However, if automated tools fail due to false positives, the only option for proving noninterference so far has been to drop the automated tools altogether and instead turn to fine-grained but interactive, and hence, more time-consuming approaches, such as theorem proving. This "all or nothing" approach is unsatisfying and problematic in practice.

We therefore propose a tool-independent hybrid approach, which allows one to use (fully) automated verification tools for checking noninterference properties (such as Joana) as much as possible and only resort to more fine-grained, but possibly interactive verification tools (typically theorem provers) at places in a program where necessary. The latter verification requires checking specific functional properties in (parts of) the program only, rather than checking the more involved noninterference properties.

The idea underlying this approach is as follows. If the verification of noninterference of a program using an automated tool fails due to (what we think are) false positives, then, following the rules of our approach, additional code is added to the program in order to make it more explicit and more clear for the automated tool that there is no illegal information flow, and by this, avoid false positives. If the automated tool now establishes that the extended program enjoys the desired noninterference property, it remains to show that the extended program is a *conservative* extension of the original program. Intuitively, this means that the additional code did not change the behavior of the original program in an essential way, including, importantly, noninterference properties. Proving that an extension is conservative requires to prove *functional* properties of (parts of) the program and will typically be carried out by an (interactive) theorem prover.

Our hybrid approach should be widely applicable—it is not tailored to specific tools or specific applications, and the basic idea is quite independent of a specific programming language.

In order to illustrate our hybrid approach, we use it to establish cryptographic privacy properties for a non-trivial Java program, namely a non-remote e-voting system, where voters submit their choices using a voting machine.

D. Slicing

The high precision of deductive verification allows for proving complex and detailed specifications for programs. This however comes at the price of manual interaction in the proof process. Therefore, in realistic programs it can happen that the verification with interactive tools becomes infeasible. At the same time, parts of the program tend to be irrelevant with respect to the specification to be proven. To remedy this, we came up with a general technique which we call *spec slicing*. The idea behind spec slicing is the following: If parts of the program do not influence the final state w.r.t. the proof obligation, they can be safely removed and function verification can be performed on the simpler program. Verification of this simpler program can then be performed without any loss of precision but with possibly much less effort. The identification and removal of irrelevant program parts can be performed by an automatic tool like Joana.

We already applied this technique to the E-Voting machine mentioned above: Parts of its implementation perform mere logging, which does not affect the voting result and therefore does not have any influence on the functional property which had to be verified. Using Joana, we gained a simpler but equivalent program without logging, which we verified using KeY to establish functional correctness for the whole program.

III. Ongoing and Future Work

Our motivating application is to provide code level, cryptographic security guarantees for a realistic and usable e-voting system. We have designed and implemented such an e-voting system, called *sElect*. In our ongoing work, we use the CVJ framework and the verification techniques described above to obtain formal, cryptographic guarantees for this system on the code level. This work also involves further improvements of the verification techniques used for proving non-interference.

Acknowledgment

This work was funded by *Deutsche Forschungsgemeinschaft* (DFG) within the Priority Program 1496 "Reliably Secure Software Systems - RS^3", project grants BE2334/6-2, KU1434/6, SN11/12-1, GZ BU 2924/1-1.

References

[1] R. Küsters, T. Truderung, and J. Graf, "A Framework for the Cryptographic Verification of Java-like Programs," in *25th IEEE Computer Security Foundations Symposium (CSF 2012)*. IEEE Computer Society, 2012, pp. 198–212.

[2] R. Canetti, "Universally Composable Security: A New Paradigm for Cryptographic Protocols," in *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science (FOCS 2001)*. IEEE Computer Society, 2001, pp. 136–145.

[3] R. Küsters, "Simulation-Based Security with Inexhaustible Interactive Turing Machines," in *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW-19 2006)*. IEEE Computer Society, 2006, pp. 309–320, see http://eprint.iacr.org/2013/025/ for a full and revised version.

[4] J. Graf, M. Hecker, and M. Mohr, "Using joana for information flow control in java programs - a practical guide," in *Proceedings of the 6th Working Conference on Programming Languages (ATPS'13)*, ser. Lecture Notes in Informatics (LNI) 215. Springer Berlin / Heidelberg, Feb. 2013, pp. 123–138.

[5] D. Wasserrab, "From Formal Semantics to Verified Slicing - A Modular Framework with Applications in Language Based Security," Ph.D. dissertation, Karlsruher Institut für Technologie, Fakultät für Informatik, Oct. 2010. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000020678

[6] W. Ahrendt, B. Beckert, D. Bruns, R. Bubel, C. Gladisch, S. Grebing, R. Hähnle, M. Hentschel, M. Herda, V. Klebanov, W. Mostowski, C. Scheben, P. H. Schmitt, and M. Ulbrich, "The KeY platform for verification and analysis of Java programs," in *Verified Software: Theories, Tools, and Experiments (VSTTE 2014)*, ser. Lecture Notes in Computer Science, D. Giannakopoulou and D. Kroening, Eds., no. 8471. Springer-Verlag, 2014, pp. 1–17. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-319-12154-3_4

[7] B. Beckert, "A dynamic logic for Java Card," in *Proceedings, 2nd ECOOP Workshop on Formal Techniques for Java Programs, Cannes, France*, 2000, pp. 111–119.

[8] G. T. Leavens, A. L. Baker, and C. Ruby, "JML: a Java Modeling Language," in *Formal Underpinnings of Java Workshop (at OOPSLA '98)*, Oct. 1998.