A Hybrid Approach for Proving Noninterference and Applications to the Cryptographic Verification of Java Programs

Ralf Küsters*, Tomasz Truderung*, Bernhard Beckert†, Daniel Bruns†, Jürgen Graf† and Christoph Scheben†

*University of Trier, Germany
Email: {kuesters,truderung}@uni-trier.de
†Karlsruhe Institute of Technology, Germany
Email: {beckert,bruns,graf,christoph.scheben}@kit.edu

Abstract—Several tools and approaches for proving noninterference properties for Java and other languages exist. Some of them have a high degree of automation or are even fully automatic, but overapproximate the actual information flow, and hence, may produce false positives. Other tools, such as those based on theorem proving, are more precise, but need more interaction, and hence, analysis is time-consuming.

In this paper, we propose an approach, which we call hybrid approach, that aims at obtaining the best of both approaches. We want to use fully automatic analysis as much as possible and only at places in a program where, due to overapproximation, the automatic approaches fail, we resort to more precise, but interactive analysis, where the latter involves merely the verification of specific functional properties in certain parts of the program, rather than checking more intricate noninterference properties for the whole program.

While this hybrid approach is applicable in all contexts in which noninterference properties need to be checked, our main motivation comes from verifying cryptographic indistinguishability properties for Java programs. In previous work, Küsters, Truderung, and Graf developed a framework by which cryptographic indistinguishability properties for a Java program can be established by checking (standard) noninterference properties for the program. The main idea of this framework is to analyze a Java program first based on ideal functionalities and then replace the ideal functionalities by their realizations, in the cryptographic sense of universal composability.

In a case study, we use the hybrid approach — along with the automatic tool Joana for checking noninterference properties for Java programs and the theorem prover KeY — and the mentioned framework to establish cryptographic privacy properties for a simple e-voting system.

In order to apply the framework in this case study, we propose an ideal functionality for secure message transmission formulated in Java and provide a modular proof for its realization in Java. This result is of independent interest and another important contribution of our paper.

I. INTRODUCTION

The problem of checking noninterference properties of programs has a long tradition in the field of computer security and, in particular, in language-based security [43]. A program is called noninterferent (w.r.t. confidentiality) if no information from high variables, which contain confidential information, flows to low variables which can be observed by the attacker or an unauthorized user. Several tools and approaches exist in the literature for checking noninterference. Some approaches, such as type checking, abstract interpretations, and program dependency graphs, with tools including Joana [27], JIF [39], TAJ [45] as well as tools described in [4] and [26], have a high degree of automation, but they overapproximate the actual information flow, and hence, may produce false positives. Others, such as those based on theorem proving, with tools such as KeY [1], Isabelle [41], and Coq [14], allow for very precise analysis, but need human interaction and, hence, analysis is often time-consuming (see, e.g., [5], [8], [9], [13], [21], [38], [40], [46], [47]).

Certainly, fully automated tools are preferable over interactive approaches. However, if automated tools fail due to false positives and the analysis cannot further be refined by these tools, because, for example, the tools do not allow this or run into scalability problems, the only option for proving noninterference so far is 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.

In this paper, we therefore propose a simple, tool-independent approach, which we call hybrid approach and which allows one to use automated analysis of noninterference properties as much as possible and only resort to more fine-grained analysis at places in a program where necessary, where the latter analysis requires checking merely specific functional properties in parts of the program, rather than checking the more involved noninterference properties (for the whole program).

While 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—, our main motivation comes from the problem of checking, on the implementation level, cryptographic properties of programs (that use cryptography), where here we consider Java programs. This has become an active field of research in the last few years (see, e.g., [22], [35], [34], [3], [2], [24], [15], [20], [17], [7], [16]). In this paper, besides the hybrid approach, we make contributions also to this
problem, which are of independent interest. More precisely, the contributions of this paper are as follows.

Contributions. Our hybrid approach is stated and proven for the language Jinja+, a rich fragment of Java. The basic idea underlying this approach is as follows: Given a program $P$, we first run an automated tool on $P$. If this fails due to (what we think are) false positives, we add some code, following rules of our approach, to $P$ at places where the tool has problems, to make it more explicit and more clear for the automated tool that there is no illegal information flow. A typical case is that we add an assignment to a variable with an expression that makes explicit that this variable does not depend on high input. (Formulating the expression may require to gather some data at other places in a program.) It might be necessary to iterate this process until the tool does not produce false positives. Let $P'$ denote the resulting extension of $P$ and assume that the automated tool showed that $P'$ has the desired noninterference property. Now, we need to show that $P'$ is what we call a conservative extension of $P$. This basically means that $P$ and $P'$ behave the same, i.e., the additional code did not change the behavior of the original program. In particular, if an assignment was added, then right before the execution of the assignment the variable should already have the value that is then assigned to it. In other words, the assignment is redundant. Proving that an extension is conservative would now typically require some more precise and possibly interactive tool. However, the analysis should typically be restricted to certain fragments of the program (namely parts in which the automated tool had problems) and involve merely the analysis of specific functional properties, rather than checking the more intricate noninterference properties (for the whole program). The key property that we show for the hybrid approach to work is that if $P'$ is noninterferent and is a conservative extension of $P$, then $P$ is noninterferent as well. To the best of our knowledge, this seems to be a new approach for proving noninterference.

As mentioned, our hybrid approach should be widely applicable. The basic concept is quite independent of a specific programming language. Also, the approach is not tailored to specific tools or applications.

The application domain for the hybrid approach we are mainly interested in is the problem of checking cryptographic indistinguishability properties for Java programs. In [35], a framework was developed that enables tools that can check (standard) noninterference properties for Java programs, but a priori cannot deal with cryptography (probabilities, polynomially bounded adversaries), to establish cryptographic indistinguishability properties of Java programs. This framework was further developed in [34]. The framework combines techniques from program analysis and universal composability [18], [42], [33]. Given a Java program (that uses cryptography), the idea is to first check noninterference for this program where cryptographic operations (such as encryption) are performed within so-called ideal functionalities, in the sense of universal composability. The framework then guarantees that the actual Java program, where the ideal functionalities are replaced by the actual cryptographic operations, enjoys cryptographic indistinguishability properties.

In a case study, we use the hybrid approach and the mentioned framework to establish cryptographic privacy properties for a simple e-voting system implemented in Java. In this system, voters can send their votes over a confidential and authenticated channel (realized using public-key encryption and signatures) to a server which, after the voting phase is finished, calculates the result of the election and posts it, using an authenticated channel (realized using signatures), on a bulletin board. Everybody can then obtain the result of the election from the bulletin board. Since we are interested in checking privacy of votes for this system, in a set-up phase, the adversary can provide two vectors of votes for honest voters. It is checked whether these vectors result in the same election outcome, i.e., whether the number of votes for each candidate is the same. If this is not the case, the system aborts. Otherwise, honest voters vote according to one of the vectors provided by the adversary. Which vector is chosen depends on a secret (high) bit. Now, the system provides privacy if the adversary cannot distinguish which of the two vectors was chosen. In other words, secrecy of the high bit is preserved, a property which, based on the mentioned framework, we would like to establish by an automated tool for checking noninterference. In our case study, we use the fully automated tool Joana [27] for this task.

The problem is that Joana produces a false positive (and probably all other automated tools would do this for our e-voting system). Joana cannot see that the publication of the election outcome does not constitute an illegal information flow. Roughly, the reason that there is no information leakage is that the election outcome is determined by the vectors provided by the adversary, which constitute low input and induce the same election outcome. So provided that the server calculates the correct result, it should in fact correspond to the result induced by the vectors. As a side remark, by using this modeling (two vectors of votes provided by the adversary), we avoid the problem of declasification and at the same time obtain a strong notion of privacy. Now, in order to see that there is no illegal flow, Joana would have to verify that the server calculates and outputs the correct result, namely the one induced by the vectors. This is beyond what Joana can do.

Using our hybrid approach, in combination with Joana and the theorem prover KeY [1], we can nevertheless establish the desired property for our system, where KeY needs to
prove only the functional property that the server correctly calculates and outputs the election result. The fact that otherwise the clients, the server, and the bulletin board do not leak secret information is established by Joana. We note that the analysis with KeY is mostly finished but is ongoing work (see Section VI-C).

In order to apply the mentioned framework to this case study — by which we obtain cryptographic guarantees by verifying (standard) noninterference properties —, we need to provide an ideal functionality for secure message transmission, i.e., confidential and authenticated message transmission, as well as an ideal functionality for authenticated message transmission, and we need to show that these functionalities can be realized using standard (IND-CCA2-secure) encryption schemes and (EU-CMA-secure) signature schemes. By the framework, it then suffices to verify noninterference of our e-voting system when it uses these functionalities instead of the actual cryptographic schemes.

These ideal functionalities and their realizations are of general interest. They can be used beyond this particular case study to establish cryptographic indistinguishability properties for Java programs that use such primitives. They also further instantiate the framework of [35], for which so far only an ideal functionality for public-key encryption has been considered. As such they constitute another important contribution of our work. The proofs of the realizations of these functionalities are non-trivial. They are carried out in a modular way. While in the cryptographic literature similar functionalities and their realization have been considered before in a Turing machine model (see, e.g., [42]), here these functionalities are formulated in Java and they can actually be used in Java programs. This requires some care. In addition, the proofs are carried out with respect to Jinja+ semantics. (We note that already in the simpler Turing machine model the design of functionalities and proofs of realizations require care, as the history of such functionalities shows, see, e.g., discussions in [6], [19], [37]).

Structure of the paper. In the next section, we briefly recall the framework from [35]. We then, in Section III, present our hybrid approach. The functionalities and their realizations are given in Section IV. The tools Joana and KeY, that we use for our case study, are briefly introduced in Section V. The case study, including the description of the e-voting system and the verification process, is presented in Section VI. Further details and proofs are provided in the appendix.

II. FRAMEWORK FOR CRYPTOGRAPHIC VERIFICATION OF JAVA PROGRAMS

We briefly recall the framework from [35]. The definitions and theorems stated here are somewhat simplified and informal, but should suffice to follow the rest of the paper. We refer the reader to [35] for full details.

As mentioned in the introduction, the framework enables tools that can check (standard) noninterference properties for Java programs, but a priori cannot deal with cryptography (probabilities, polynomially bounded adversaries), to establish cryptographic indistinguishability properties of Java programs. To this end, the framework combines techniques from program analysis and universal composability [18], [42], [33]. Given a Java program that uses cryptographic operations, the framework shows that in order to verify that the program enjoys a cryptographic indistinguishability property it suffices to prove, using the tools, that the program enjoys a (standard) noninterference property when the cryptographic operations are replaced by so-called ideal functionalities.

The reason for using ideal functionalities is that they often do not involve probabilistic operations and are secure even for unbounded adversaries which is the kind of adversaries considered for standard noninterference properties.

Jinja+. The framework is stated and proven for a Java-like language called Jinja+. Jinja+ is based on Jinja [32] and extends this language with some additional features that are useful or needed in the context of our framework.

Jinja+ covers a rich subset of Java, including classes, inheritance, (static and non-static) fields and methods, the primitive types int, boolean, and byte (with the usual operators for these types), arrays, exceptions, and field/method access modifiers, such as public, private, and protected. It also includes a primitive randomBit() that returns a random bit each time it is called.

A (Jinja+) program/system is a set of class declarations. A class declaration consists of the name of the class, the name of its direct superclass, a list of field declarations, and a list of method declarations. A program/system is complete if it uses only classes/methods/fields declared in the program itself.

All Java programs considered in this paper, including the systems considered in our case study as well as the ideal functionalities and their realizations fall into the Jinja+ fragment.

Indistinguishability. An interface $I$ is defined like a (Jinja+) system but where (i) all private fields and private methods are dropped and (ii) method bodies as well as static field initializers are dropped. A system $S$ implements an interface $I$, written $S : I$, if $I$ is a subinterface of the public interface of $S$. We say that a system $S$ uses an interface $I$, written $I :: S$, if, besides its own classes, $S$ uses at most classes/methods/fields declared in $I$. We write $I_0 :: S : I_1$ for $I_0 :: S$ and $S : I_1$.

For two system $S$ and $T$ we denote by $S : T$ the composition of $S$ and $T$ which, formally, is the union of (declarations in) $S$ and $T$. Clearly, for the composition to make sense, we require that there are no name clashes in the declarations
of $S$ and $T$. Of course, $S/T$ may use classes/methods/fields provided in the public interface of $T/S$.

A system $E$ is called an environment if it declares a distinct private static variable result of type boolean with initial value false. Given a system $S : I$, we call $E$ an $I$-environment for $S$ if there exists an interface $I_E$ disjoint from $I$ such that $I_E \vdash S : I$ and $I \vdash E : I_E$. Note that $E \cdot S$ is a complete program. The value returned by $E$ to result at the end of a run of $E \cdot S$ is called the output of the program $E \cdot S$; the output is false for infinite runs. If $E \cdot S$ is a deterministic program, we write $E \cdot S \overset{\text{true}}{\rightarrow}$ if the output of $E \cdot S$ is true. If $E \cdot S$ is a randomized program, we write $\text{Prob}\{E \cdot S \overset{\text{true}}{\rightarrow}\}$ to denote the probability that the output of $E \cdot S$ is true.

We assume that all systems have access to a security parameter (modeled as a public static variable of a class SP). We denote by $P(\eta)$ a program $P$ running with security parameter $\eta$.

To define computational equivalence and computational indistinguishability between (probabilistic) systems, we consider systems that run in (probabilistic) polynomial time in the security parameter. We omit the details of the runtime notions used in the framework, but note that the runtimes of systems and environments are defined in such a way that their composition results in polynomially bounded programs.

Let $P_1$ and $P_2$ be (complete, possibly probabilistic) programs. We say that $P_1$ and $P_2$ are computationally equivalent, written $P_1 \equiv_{\text{comp}} P_2$, if $|\text{Prob}\{P_1(\eta) \overset{\text{true}}{\rightarrow}\} - \text{Prob}\{P_2(\eta) \overset{\text{true}}{\rightarrow}\}|$ is a negligible function in the security parameter $\eta$.

Let $S_1$ and $S_2$ be probabilistic polynomially bounded systems. Then $S_1$ and $S_2$ are computationally indistinguishable w.r.t. $I$, written $S_1 \approx_{\text{comp}}^{I} S_2$, if $S_1 : I$, $S_2 : I$, both systems use the same interface, and for every polynomially bounded environment $E$ for $S_1$ (and hence, $S_2$) we have that $E \cdot S_1 \equiv_{\text{comp}} E \cdot S_2$.

Simulatability and Universal Composition. We now define what it means for a system to realize another system, in the spirit of universal composability. Security is defined by an ideal system $F$ (also called an ideal functionality), which, for instance, models ideal encryption, signatures, MACs, key exchange, or secure message transmission. A real system $R$ (also called a real protocol) realizes $F$ if there exists a simulator $S$ such that no polynomially bounded environment can distinguish between $R$ and $S \cdot F$. The simulator tries to make $S \cdot F$ look like $R$ for the environment (see Section IV for examples).

More formally, let $F$ and $R$ be probabilistic polynomially bounded systems which implement the same interface $I_{\text{out}}$ and use the same interface, except that in addition $F$ may use some interface provided by a simulator. Then, we say that $R$ realizes $F$ w.r.t. $I_{\text{out}}$, written $R \leq_{I_{\text{out}}} F$ or simply $R \leq F$, if there exists a probabilistic polynomially bounded system $S$ (the simulator) such that $R \approx_{\text{comp}}^{S} F$. As shown in [35], $\leq$ is reflexive and transitive.

A main advantage of defining security of real systems by the realization relation $\leq$ is that systems can be analyzed and designed in a modular way because of the following composition theorem.

**Theorem 1 (Composition Theorem (simplified) [35]).** Let $I_0$ and $I_1$ be disjoint interfaces and let $R_0$, $F_0$, $R_1$, and $F_1$ be probabilistic polynomially bounded systems such that $R_0 \leq_{I_0} F_0$ and $R_1 \leq_{I_1} F_1$. Then, $R_0 \cdot R_1 \leq_{I_0\cup I_1} F_0 \cdot F_1$.

The theorem implies that it suffices to prove security separately for $R_0$ and $R_1$ in order to obtain security of the composed system.

**Noninterference.** The (standard) noninterference notion for confidentiality [23] requires the absence of information flow from high to low variables within a program. Here, we define noninterference for a deterministic (Jinja+) program $P$ with some static variables $\bar{x}$ of primitive types that are labeled as high. Also, some other static variables of primitive types are labeled as low. We say that $P[\bar{x}]$ is a program with high and low variables. By $P[\bar{a}]$, we denote the program $P$ where the high variables $\bar{x}$ are initialized with values $\bar{a}$ and the low variables are initialized as specified in $P$.

Now, noninterference for a deterministic program is defined as follows: Let $P[\bar{x}]$ be a program with high and low variables. Then, $P[\bar{x}]$ has the noninterference property if the following holds: for all $\bar{a}_1$ and $\bar{a}_2$ (of appropriate type), if $P[\bar{a}_1]$ and $P[\bar{a}_2]$ terminate, then at the end of their runs, the values of the low variables are the same. Note that this defines termination-insensitive noninterference.

The above notion of noninterference deals with complete programs (closed systems). This notion is generalized to open systems as follows: Let $I$ be an interface and let $S[\bar{x}]$ be a (not necessarily closed) deterministic system with a security parameter and high such that $S : I$. Then, $S[\bar{x}]$ is $I$-noninterferent if for every deterministic $I$-environment $E$ for $S[\bar{x}]$ and every security parameter $\eta$, noninterference holds for the system $E \cdot S[\bar{x}] (\eta)$, where the variable result declared in $E$ is considered to be the only low variable. Note that here neither $E$ nor $S$ are required to be polynomially bounded.

Tools for checking noninterference often consider only a single closed program. However, $I$-noninterference is a property of a potentially open system $S[\bar{x}]$, which is composed with an arbitrary $I$-environment. Therefore in [35] a technique was developed which reduces the problem of checking $I$-noninterferent to checking noninterference for a single (almost) closed system. More specifically, it was...
shown that to prove $I$-noninterference for a system $S[\vec{x}]$ with $I_E \vdash S : I$ it suffices to consider only a single environment $E_{\vec{u}} = E_{\vec{u}}^{I_E}$, which is parameterized by a sequence $\vec{u}$ of values. The output produced by $E_{\vec{u}}$ to $S[\vec{x}]$ is determined by $\vec{u}$ and is independent of the input it gets from $S[\vec{x}]$. To keep $\vec{u}$ simple, the analysis technique assumes some restrictions on interfaces between $S[\vec{x}]$ and $E$. For example, $S[\vec{x}]$ and $E$ should interact only through primitive types, arrays, exceptions, and simple objects.

**Theorem 3** (simplified, [35]). $I$-noninterference, for $I = \emptyset$, holds for $S[\vec{x}]$ if and only if for all sequences $\vec{u}$ noninterference holds for $E_{\vec{u}} : S[\vec{x}]$.

Analysis tools often ignore or can ignore specific values encoded in a program, such as an input sequence $\vec{u}$. So, if such an analysis establishes noninterference for $E_{\vec{u}} : S[\vec{x}]$, with some vector $\vec{u}$, the theorem implies $I$-noninterference for $S[\vec{x}]$.

From $I$-Noninterference to Computational Indistinguishability. In [35], it was shown that the problem of verifying cryptographic privacy of the secret (high) input given to a system $S[\vec{x}]$ can be reduced to the simpler problem of verifying $I$-noninterference for $S[\vec{x}]$ where the cryptographic operations performed by $S[\vec{x}]$ are replaced by ideal functionalities. This enables tools that can check (standard) noninterference properties but cannot deal with cryptography (probabilities, polynomially bounded adversaries) to establish strong cryptographic privacy properties. Note that such tools assume unbounded adversaries, which can easily break basically all cryptographic operations, such as encryption. The result in [35] avoids this problem by replacing the cryptographic operations by ideal functionalities, which are secure even w.r.t. unbounded adversaries. More specifically, the following theorem immediately follows from results proven in [35].

**Theorem 2** (simplified, [35]). Let $I$ and $J$ be disjoint interfaces. Let $F$, $R$, $P[\vec{x}]$ be systems such that $R \leq^I F$, $P[\vec{x}] \cdot F$ is deterministic, and $P[\vec{x}] \cdot F : I$ (and hence, $P[\vec{x}] \cdot R : I$). Now, if $P[\vec{x}] \cdot F$ is $I$-noninterferent, then, for all $\vec{a}_1$ and $\vec{a}_2$ (of appropriate type), we have that $P[\vec{a}_1] \cdot R \approx^I_{\text{comp}} P[\vec{a}_2] \cdot R$.

The intuition is that the cryptographic operations that $P$ needs to perform are carried using the system $R$ (a cryptographic library). The theorem says that to prove privacy of the secret inputs ($\forall \vec{a}_1$, $\vec{a}_2$: $P[\vec{a}_1] \cdot R \approx^I_{\text{comp}} P[\vec{a}_2] \cdot R$) it suffices to prove $I$-noninterference for $P[\vec{x}] \cdot F$, i.e., the system where $R$ is replaced by the ideal counterpart $F$.

---

The result in [35] is actually more general and considers general cryptographic indistinguishability properties, but here we merely need privacy properties.

### III. A Hybrid Approach for Proving Noninterference

In this section, we present our hybrid approach. This approach is a way to leverage the precision of certain tools, such as theorem provers (e.g., KeY), in order to allow less precise, but automated tools, such as Joana, to prove noninterference of programs which otherwise they would not be able to deal with. More specifically, our approach is useful when we want to use a highly automated tool to prove noninterference of a given program $P$, but the tool reports (as we believe) a false positive, i.e., illegal information flow in a point of a program where there is no such illegal flow. If the analysis of the automated tool cannot further be refined, the only option so far for proving noninterference is to give up completely on the automated tool(s) and try to use a more precise, but interactive tool, which, however, typically would involve a substantial, potentially unacceptable amount of work. Our hybrid approach avoids this problem. It allows one to still use the automated tool as much as possible and resort to other more interactive tools only at places in the program where necessary, namely where the automated tool failed.

In a nutshell, our hybrid approach works as follows. Given a program $P$ for which we want to verify noninterference properties, we would first run the automated tool on $P$. If this fails due to (what we think are) false positives, we would add some code, following rules of our approach, to $P$ at places where the tool has problems, to make it more explicit and more clear for the automated tool that there is no illegal information flow. We would then run the automated tool on the extended program again. If still there is a problem (potentially at other places in the program), we would again add code, and so on, until the automated tool does not produce false positives anymore. Let $P'$ denote the extension of $P$ for which this is the case. Now, we have to prove that $P'$ is what we call a conservative extension of $P$, i.e., the additional code does not change the behavior of $P$, and in particular, if $P'$ enjoys the noninterference property, then so does $P$. The property of being a conservative extension is, as we will see below, a specific functional property (of some part of the overall program). Proving that $P'$ is a conservative extension is now done using a more precise, but possibly interactive tool, e.g., a theorem prover. However, this should typically involve only analyzing a relatively small fraction of the overall program and being a functional property is easier to prove than noninterference properties (of the overall program). Also, as illustrated by the example at the end of this section and our case study, these or similar functional properties would typically have to be proven anyway when proving noninterference, because they are part of the reason that there is no information flow. Altogether, by this the potentially interactive, and hence, time-consuming part of the analysis of $P$ should be mainly restricted to the
points of the program that the automated tool is not able to handle. In particular, compared to a full-fledged analysis of noninterference of the complete program \( P \) by an interactive tool, the hybrid approach can save quite some effort and time.

To explain our hybrid approach in more detail, we start with the definition of a conservative extension.

**Definition 1.** Let \( P \) be a program. An extension of \( P \) is a program \( P' \) obtained from \( P \) in the following way. First, a new class \( M \) with the following properties is added to \( P \):

(i) the methods and fields of \( M \) are static,
(ii) the methods of \( M \) do not call any methods defined in \( P \) (that is \( \emptyset \vdash M \)),
(iii) the arguments and the results of the methods of \( M \) are of primitive types,
(iv) all potential exceptions are caught inside \( M \).

Second, \( P \) is extended by adding statements of the following form in arbitrary places within methods of \( P \):

(a) (output to \( M \))

\[
M.f(e_1,\ldots,e_n),
\]

where \( f \) is a (static) method of \( M \) and \( e_1,\ldots,e_n \) are expressions without side effects and of the types corresponding to the types of the arguments of \( M.f \).

(b) (input from \( M \))

\[
r = M.f(e_1,\ldots,e_n),
\]

where \( f \) is a method of \( M \) with some (primitive) return type \( t \), expressions \( e_1,\ldots,e_n \) are as above, and \( r \) is an expression that evaluates without side-effects to a reference of type \( t \). (Such an expression can for example be a variable or an expression of the form \( o.x \), where \( o \) is an object with field \( x \).)

Such an extension \( P' \) of \( P \) is called a conservative extension of \( P \), if, additionally, the following is true. For a run of the program \( P' \), whenever a statement of the form (2) is executed, it does not change the value of \( r \). That is, the value of \( r \) right before the execution of the assignment coincides with the value returned by the method call \( M.f(e_1,\ldots,e_n) \).

As such, statement (2) is redundant.

The intuition behind the above definition is the following. First, the properties of \( M \) guarantee that the state of \( M \) is disjoint from the state of the original program \( P \) (i.e. these components do not share any references). In particular, statements of the form (1) preserve the state of \( P \). Second, the additional properties of a conservative extension guarantee that the assignments of the form (2) also do not change the state of \( P \). Altogether, \( P \) and its conservative extension \( P' \) are equivalent in that they produce the same runs up to calls to \( M \) and up to the state of \( M \).

The property of being an extension of a program is a rather simple condition. Conditions (i) to (iii) can easily be checked syntactically. As for Condition (iv), typically \( M \) should be a quite simple class, and hence, it is should be easy to see that methods cannot throw exceptions. If there are potentially problematic operations, one can explicitly catch them inside \( M \). Whether this is done can, again, be checked syntactically.

Proving that the extension is conservative, i.e., adding statements of the form (2) does not change the state of a program, requires a (possibly non-trivial) proof. For this one would use some potentially interactive tool.

The definition of conservative extensions given above can be generalized as follows (see Appendix A for details). First, we can allow the arguments of methods of \( M \) to be references to objects, such as arrays. (This extends (iii) in the above definition.) However, in this case we require that \( M \) first makes a copy of the data of the object and then no longer uses the reference. By this, \( M \) basically does not share any reference with \( P \), and hence, \( M \) cannot influence the state of \( P \) by manipulating references. Second, (2) in the above definition could be generalized to allow for adding arbitrary code in \( P \) that does not change the state of \( P \) itself, i.e., in every run before and after this piece of code, \( P \) is in the same state. Again, for verifying that the code added does in fact not change the state of \( P \), one would use an (interactive) tool.

Now, following the intuition that the runs of a program \( P \) and a conservative extension of \( P \) produce the same runs up to calls to \( M \) and the state of \( M \), the following theorem can be proven, also for the generalized notion of conservative extensions sketched above (see the appendix). The theorem says that if a conservative extension of a program with high and low variables is noninterferent, then so is the program itself. Note that part of the proof of the theorem is that the conditions for \( M \) in Definition 1 in fact guarantee that \( M \) does not change the state of a program.

**Theorem 4.** Let \( P[x] \) be a deterministic program with the variables \( x \) labeled as high (and, possibly, some other variables labeled as low). If a conservative extension \( P'[x] \) of \( P[x] \) is noninterferent, then so is \( P[x] \).

The definitions and results stated above do not say how one can come up with a conservative extension of a program. This process is not automatic and requires some understanding of the analyzed program and the automated tool that is used. (However, checking noninterference properties for the whole program, rather than simpler functional properties for parts of the program, with an interactive tool would typically require much more understanding of the program and human interaction.) As sketched at the beginning of this section, the approach to come up with a conservative extension of a program is as follows: First the automated tool is run on the program. Now, assume that it reports an illegal information flow at some point of the program, of which we think that
this is a false positive. For the sake of the discussion, say that some potential high information flows into some variable \( v \) and from this point this information flows to some low variables. If the value assigned to \( v \) does not actually depend on high variables, the conservative extension should make this more explicit. This is done by giving the class \( M \) all necessary low data in order to compute the value for \( v \). This is done by adding statements of the form (1) in Definition 1 (output to \( M \)). It should be apparent for the automated tool that the data given to \( M \) is low. Then, at the problematic point in the program (typically after the point in the original program where \( v \) is assigned a value) an assignment to \( v \) of the form (2) in Definition (2) (input from \( M \)) is added to the program to make it more explicit for the automated tool that \( v \) depends only on low data. In other words, the assignment explicitly “kills” potential dependencies from high data.

In what follows, we illustrate, by a toy example, the process of running into a false positive with an automated tool, extending the code, proving that this extension has the desired noninterference property, and proving that the extension is conservative (by some interactive tool). The example also demonstrates that the hybrid approach can be very beneficial, compared to giving up on automated analysis completely and switching to an interactive tool for checking noninterference for the whole program. In Section VI, a more interesting and more complex example is provided.

To ease the notation, we will allow fields \( x \) of the class \( M \) in a conservative extension to be used in the following way:

\[
M.x = e \quad \text{(output to } M) \]

and

\[
r = M.x \quad \text{(input from } M) \]

These statements can be rewritten to the corresponding statements of the form (1) and (2) by adding appropriate getter and setter methods to the class \( M \).

**Example.** We consider the following program, where \( secret \) is declared to be a high variable and \( result \) is low.

```java
class Example {
    static public int result;
    static private int a;
    public void main(int secret) {
        a = 42;
        bar(secret);
        int b = foo(secret);
        result = b;
    }
    static int foo(int secret) {
        int b = a;
        // M.a = a
        if (secret==0) b+=secret;
        // b = M.a;
        return b;
    }
    static void bar(int secret) {
        ...
    }
}
```

We applied the fully automated tool Joana for checking noninterference to this example (see Section V-A for more information about Joana), where \( bar \) was some piece of code that Joana could handle (see [11]). Joana reported in method \( foo \), line 13 a potential information flow from variable \( secret \) to variable \( b \), the value of which is then returned and assigned to the low variable \( result \). This flow is, however, not real, as can easily be seen: the value of \( b \) is in fact not altered in line 13 and it only depends on the low variable \( a \). We can easily make this dependence explicit by adding the class \( M \) with only one static field \( a \) of type \( int \) and uncommenting lines 12 and 14. Now, Joana establishes noninterference for this extended program without problems. Note that, in particular, Joana tells us that the method \( bar \) does not cause \( a \), and hence, \( result \) to depend on high information.

It remains to prove that the extension is conservative, i.e., that before executing line 14 the variable \( b \) carries the value \( M.a (= a) \). This is merely a functional property and can be easily verified using the theorem prover KeY (see [11] for the analysis of this property with KeY and Section V-B for more background on KeY). We note that this requires only local (functional) reasoning. There is no need to inspect the whole program. In particular, the method \( bar \), which potentially might be very big and it might be a real burden to verify using a theorem prover, does not have to be inspected by KeY.

Now, by the results of Joana and KeY, Theorem 4 implies that our original example program (with lines 12 and 14 commented out) enjoys the noninterference property, a fact that with Joana alone could not have been established. It could have been established with KeY only (this tool can be used to not only verify functional but also noninterference properties), but this would have required to check noninterference for the whole program, rather than checking a simple functional property for a small part of the program.

IV. FUNCTIONALITIES FOR SECURE AND AUTHENTICATED MESSAGE TRANSMISSION

In this section, we propose ideal functionalities for secure message transmission (SMT) as well as authenticated message transmission (AMT), formulated in Java (Jinja+). We show, based on the framework proposed in [35] (see Section II) that these functionalities can be realized using encryption and signature schemes with standard security assumptions. The realizations are again formulated in Java. The proof that the realizations in fact realize the corresponding ideal functionalities are modular in that the realizations themselves use ideal functionalities for public-key encryption and digital signatures.

Similar functionalities as the one introduced in this work have been considered in the cryptographic literature based on
Turing machine models (see, e.g., [42], [18], [37]). The new contribution here is that we provide a formulation in Java, such that these functionalities can actually be used in Java programs. Designing such functionalities and carrying out the proofs (w.r.t. Jinja+ semantics) is non-trivial and requires some care since the interaction between different classes is much more complex than between Turing machines, where in the former case we have to deal, for example, with exceptions, inheritance, references to potential complex objects that can be exchanged, and hence, manipulation of one object can effect many other objects.

As already mentioned in the introduction, the functionalities for secure and authenticated message transmission are used in our case study in Section VI in order to establish, based on the framework in [35] (see Section II), cryptographic privacy properties for a Java program. They are of general interest as they enrich the collection of functionalities that have been developed for the framework so far and can be used to establish cryptographic guarantees for arbitrary Java programs that employ such primitives. Before only a public-key functionality has been devised for the framework [35].

Our functionality (and its realization) for secure message transmission enables an agent \( A \) to send a message \( m \) to an agent \( B \) such that the message transmission is authenticated and confidential, i.e., \( B \) can be sure that \( m \) originated from \( A \) and is intended for \( B \) and that nobody but \( A \) and \( B \) get to know the message. The functionality for authenticated message transmission is similar, except that it does not provide confidentiality. Note that these functionalities do not guarantee that messages are delivered in order and at most once, unlike a secure channel functionality.

In the rest of this section, we first provide the interface for the SMT functionality (and its realization). Then, the actual ideal SMT functionality and its realization are presented, along with a proof that shows that the realization actually realizes the ideal SMT functionality. The proof is modular in that it uses two other ideal functionalities, a public-key encryption and a digital signature functionality. We then briefly sketch our result on authenticated message transmission.

A. The Interface for SMT

The interface \( I_{SMT} \) of the secure message transmission functionality is the following:

```java
public class SMT {
    static SMT extends Exception {
        static public class SMTError extends Exception {
            }
        }
    }
    static public classAuthenticatedMessage {
        public byte[] message;
        public int sender_id;
    }
    static public class AgentProxy {
        public final int ID;
        public AuthenticatedMessage getMessage(int port)
    }
    public Channel channelTo( recipient_id,
        String server_id, int port)
        throws SMTError, PKIError, NetworkError;
    }
    static public class Channel {
        public void send(byte[] message);
    }
    public static AgentProxy register(int id)
        throws SMTError, PKIError;
    }
}
```

The intended usage of this interface is as follows. A party \( A \) who wants to use it needs first to register her ID, using the method \( SMT.register \):

```java
SMT.AgentProxy a = SMT.register(ID_OF_A);
```

Such a call throws PKIError if this identifier has been registered already. Otherwise, if the registration went smoothly, the returned object encapsulates all the data necessary for \( A \) to receive messages and send messages in a confidential and authenticated manner.

To read messages from the network that have been sent to \( A \), one does the following:

```java
SMT.AuthenticatedMessage msg = a.getMessage(port);
```

Now, \( msg \) may be \( null \). This means that no (valid confidential and authenticated) message to \( A \) is waiting on the network interface. Otherwise, \( msg.message \) contains a message and \( msg.sender_id \) contains the identifier of the sender of this message. It is guaranteed that \( msg.sender_id \) actually sent \( msg.message \) to \( A \) at some point. (Note, however, that one such message might be delivered multiple times to \( A \).)

To send messages to a party \( B \) with identifier ID_OF_B, the agent \( A \) first runs the following code:

```java
SMT.Channel channel_to_b = a.channelTo(ID_OF_B, serv, port);
```

The method \( channelTo \) throws an exception if no agent with the given identifier has been registered. Otherwise, a channel is created that encapsulates all the data necessary to send messages from \( A \) to \( B \) (it can be used to send messages multiple times). For example, \( A \) can run the code

```java
channel_to_b.send(message1);
channel_to_b.send(message2);
```

to send messages \( message1 \) and \( message2 \) to \( B \).

B. The Ideal Functionality for SMT

The ideal functionality for SMT, denoted by \( SMT_{Ideal} \), implements the interface \( I_{SMT} \) and the guarantees sketched above in an ideal way.

It maintains a list of objects of class AgentProxy representing registered agents. Such an object (an AgentProxy object) is created and added to this list when the agent registers herself successfully; the adversary (simulator) is informed about a registration request. An AgentProxy object contains an (initially empty) list of messages that were
sent (but not necessarily delivered) to this agent along with identifiers of the senders.

If an agent $A$ uses her proxy object (of class `AgentProxy`) to obtain a channel to an agent $B$ registered under a given identifier, the proxy of $B$ is fetched (based on the identifier of $B$) and the returned channel encapsulates a reference to this object and the identifier of $A$. When such a channel is used to send a message $m$, this message along with the identity of $A$ is added to the list of messages in the `AgentProxy` object of $B$. In addition, the adversary (simulator) is informed that agent $A$ wants to send a message of length $|m|$ to $B$. (Note that the adversary only learns the length of $m$, not $m$ itself.)

When $B$ reads a message using method `getMessage` (of his proxy), the environment is asked to provide an index into the list of messages that have been sent to $B$. The message with the corresponding index is then delivered to $B$. It may happen that no message is returned if there is no message with the given index. Note that a message when delivered is not deleted from the list stored in the `AgentProxy` object of $B$, modeling that the adversary can deliver the same message several times.

We note that this functionality so far does not handle corruption. So corruption should be dealt with outside of this functionality. While, following the cryptographic literature, it is easy to add support for corruption, we, at this point, refrained from including it to make the functionality easier to manage by analysis tools. We leave an extension to future work.

More details of this implementation are presented in Appendix B (see [11] for the full code).

C. The Realization of SMT

The basic cryptographic construction for realizing the interface $I_{\text{SMT}}$ is to first sign a message (using the private key of the sender), along with the identifier of the intended recipient of the message, and then to encrypt the message and the signature under the public-key of the recipient.

More precisely, the real protocol, denoted by $SMT_{\text{Real}}$, for secure message transmission has of course the same public interface $I_{\text{SMT}}$ as the ideal protocol $SMT_{\text{Ideal}}$. The system $SMT_{\text{Real}}$ is a composition of the classes $SMT$, $PKI_{\text{Enc}, \text{Real}}$, and $PKI_{\text{Sig}, \text{Real}}$, i.e., $SMT_{\text{Real}} = SMT \cdot PKI_{\text{Enc}, \text{Real}} \cdot PKI_{\text{Sig}, \text{Real}}$. The class $SMT$ uses the class $PKI_{\text{Enc}, \text{Real}}$ for public-key encryption and the class $PKI_{\text{Sig}, \text{Real}}$ for digital signatures. The latter two classes internally use classes for a public-key infrastructure that we provide, where parties can register their public encryption and verification keys, respectively. Hence, in order to encrypt a message, using $PKI_{\text{Enc}, \text{Real}}$, or verify a signature, using $PKI_{\text{Sig}, \text{Real}}$, it suffices to provide the identity of the party in question. Encryption/decryption and signing/verifying are done in $PKI_{\text{Enc}, \text{Real}}$ and $PKI_{\text{Sig}, \text{Real}}$, respectively, using a Java cryptographic library (where encryption is assumed to be IND-CCA2-secure and signing to be EU-CMA-secure, see below).

The method $\text{SMT}\_\text{register}(\text{id})$ in $SMT$ is implemented by registering $\text{id}$ both in $PKI_{\text{Enc}, \text{Real}}$ and $PKI_{\text{Sig}, \text{Real}}$. The resulting `AgentProxy` object encapsulates the secret keys of the agent $\text{id}$, that is, the decryption key and the signing key of that agent. Messages are sent to an agent using encryption and signing as sketched above.

More details about this implementation are presented in Appendix B (see [11] for the full code, including the code for $PKI_{\text{Enc}, \text{Real}}$ and $PKI_{\text{Sig}, \text{Real}}$).

D. Realization Result

In this section, we state the realization theorem for the SMT functionality, i.e., that $SMT_{\text{Real}}$ realizes $SMT_{\text{Ideal}}$ w.r.t. the interface $I_{\text{SMT}}$, in the sense introduced in Section II.

Theorem 5. $SMT_{\text{Real}} \leq I_{\text{SMT}} SMT_{\text{Ideal}}$.

The proof of this theorem is modular and is given in Appendix B. Here we only sketch it.

As explained before, the structure of the real protocol $SMT_{\text{Real}}$ is $SMT_{\text{Real}} = SMT \cdot PKI_{\text{Enc}, \text{Real}} \cdot PKI_{\text{Sig}, \text{Real}}$. Following the universal composability paradigm, to perform a modular proof, we first consider ideal functionalities for encryption and signing, $PKI_{\text{Enc}, \text{Ideal}}$ and $PKI_{\text{Sig}, \text{Ideal}}$, respectively, such that $PKI_{\text{Enc}, \text{Ideal}}$ realizes $PKI_{\text{Enc}, \text{Real}}$ (w.r.t. the interface $I_{\text{PKI_{Enc}}}$) and $PKI_{\text{Sig}, \text{Real}}$ realizes $PKI_{\text{Sig}, \text{Ideal}}$ (w.r.t. the interface $I_{\text{PKI_{Sig}}}$), under standard cryptographic assumptions, i.e., $PKI_{\text{Enc}, \text{Real}}$ implements an IND-CCA2-secure encryption scheme and that $PKI_{\text{Sig}, \text{Real}}$ implements an EU-CMA-secure signature scheme (see [11] for the full code of $PKI_{\text{Enc}, \text{Ideal}}$ and $PKI_{\text{Sig}, \text{Ideal}}$). These realization results follow directly from [34], where stronger realization results are established (the functionalities considered in [34] are very similar, but, additionally, model static corruption).

Now, by applying the composition theorem (Theorem 1) we immediately conclude that $PKI_{\text{Enc}, \text{Real}} \cdot PKI_{\text{Sig}, \text{Real}} \leq I_{\text{PKI_{Enc}, PKI_{Sig}}} PKI_{\text{Enc}, \text{Ideal}} \cdot PKI_{\text{Sig}, \text{Ideal}}$. By reflexivity of $\leq$, we know that $SMT$ realizes $SMT$; and hence, by applying the composition theorem again, we immediately obtain that $SMT$ using the real encryption and signing operations realizes $SMT$ using ideal encryption and signing. More specifically, let $SMT^\ast = SMT \cdot PKI_{\text{Enc}, \text{Ideal}} \cdot PKI_{\text{Sig}, \text{Ideal}}$, then we have:

Lemma 1. $SMT_{\text{Real}} \leq I_{\text{SMT}} SMT^\ast$.

To finish the proof of Theorem 5, it suffices to prove the following lemma.

Lemma 2. $SMT^\ast \leq I_{\text{SMT}} SMT_{\text{Ideal}}$.

Indeed, Theorem 5 now follows from Lemma 1 and Lemma 2 by transitivity of $\leq$. 


We note that the proof of Lemma 2 does not involve any cryptographic reasoning anymore because SMT uses ideal functionalities for public-key encryption and signing.

To prove Lemma 2, we need to provide a simulator, which we denote by SMTSim, and show that $\text{SMT}^* \approx_{\text{comp}} \text{SMTSim}$, i.e., for all (polynomially bounded) $\text{SMT}$-environments $E$ we have that $E \cdot \text{SMT}^* \equiv_{\text{comp}} E \cdot \text{SMTSim} \cdot \text{SMT}^\text{ideal}$. We in fact show this for all unbounded and deterministic $\text{SMT}$-environments $E$ the output of $E$ is the same on both sides (this is called perfect indistinguishability and is shown to imply computational indistinguishability in [35]).

The simulator $\text{SMTSim}$ is defined as follows (see Appendix B-E for more details and [11] for the full Java code for $\text{SMTSim}$).

The simulator maintains its own copy of $\text{SMT}^*$, i.e., part of the code of $\text{SMTSim}$ is $\text{SMT}^*$ (up to some renaming and some small changes). In addition, $\text{SMTSim}$ does some bookkeeping and interacts with $\text{SMT}^\text{ideal}$ when necessary. Whenever a method of $\text{SMT}^\text{ideal}$ is called (via the interface $\text{ISMT}$) and the simulator is notified, $\text{SMTSim}$ calls the corresponding method of the copy of $\text{SMT}^*$. The simulator collects all the objects returned by its copy of $\text{SMT}^*$ and, additionally, maintains some data structures that allow it to retrieve appropriate objects based on user identifiers.

Whenever a channel object of $\text{SMT}^\text{ideal}$ representing a channel from $A$ to $B$ is used to send a message $m$, the simulator fetches the appropriate channel object of the simulated copy of $\text{SMT}^*$ and calls its send method. As a result, it obtains the message $c$ that this method sends over the network. The simulator stores $c$, so that it can later, when this message is delivered to $B$ as dictated by the environment, reconstruct the corresponding index in the list of messages sent to $B$ maintained by $\text{SMT}^\text{ideal}$.

With this simulator, we show that, given a deterministic $\text{ISMT}$-environment $E$, in the run of the (deterministic) system $E \cdot \text{SMT}^*$ strictly correspond to the states of the simulated copy of $\text{SMT}^*$ in the (deterministic) system $E \cdot \text{SMTSim} \cdot \text{SMT}^\text{ideal}$. Based on this, we show that the corresponding runs are exactly the same from the point of view of the environment. In other words, we show that $\text{SMT}^*$ and $\text{SMT}^\text{ideal} \cdot \text{SMTSim}$ are perfectly indistinguishable (and hence, computationally indistinguishable).

E. Authenticated Message Transmission

We also provide an ideal functionality $\text{AMT}^\text{ideal}$ for authenticated message transmission (AMT) along with a realization $\text{AMT}^\text{real}$. As mentioned, the only difference compared to SMT is that now merely the authenticity of messages is guaranteed, but no confidentiality. That is, in $\text{AMT}^\text{ideal}$ the adversary (simulator) is given the actual message, rather than only the length of a message. For the realization, we can dispense with the encryption of messages. Now, similarly to the proof of Theorem 5 and again assuming that the signature scheme used for the realization is EU-CMA-secure, we obtain:

**Theorem 6.** $\text{AMT}^\text{real} \leq_{\text{AMT}} \text{AMT}^\text{ideal}$

V. Tools

In our case study (and in the toy example from Section III) we use the tools Joana and KeY. We provide some background about these tools in this section.

A. Joana

Joana$^3$ [28], [25] is a tool for the fully automatic analysis of noninterference properties of Java programs. A user only needs to specify the high and low variables of a program. Joana is based on the program analysis framework WALA$^4$. It computes a conservative approximation of the information flow inside the program in form of a program dependence graph (PDG). Then, the PDG is checked for illegal information flow 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 [48], that includes formal specifications of PDGs and the slicing algorithm [49], [50].

The fully automatic analysis performed by Joana comes at the cost of potential false alarms due to over-approximation. Joana leverages sophisticated flow-, context-, field- and object-sensitive analysis techniques that help to reduce false alarms, but it does not consider actual values of variables. For example, whenever a high variable is used in an expression, the value of the expression is considered to contain high information – even if the value of the high variable does not actually influence the result (this phenomenon is also illustrated by the example in Section III and our case study in Section VI).

B. The KeY Tool

KeY$^5$ [1], [12] is a program verification system, which targets sequential Java. At its core is an interactive theorem prover for first-order dynamic logic (JavaDL) [29], [10], a super-set of Hoare logic [30]. In contrast to Hoare triples, however, programs are part of formulae. This allows one to write down more elaborate formulae, e.g., formulae with multiple programs or existential quantification ranging over program states. In particular, information flow properties such as noninterference can be expressed [44] (a feature that, however, in the scope of this work is not needed).

3The sourcecode of Joana and additional information is available at http://joana.ipd.kit.edu/.
4http://wala.sf.net/
5KeY is free software and can be downloaded (in source or binary) from http://key-project.org/, the current stable version is 2.0.
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 (both functional and information flow) built on KeY are precise. They do not report any false positives.

KeY is different from general purpose (first-order or higher-order) theorem provers (such as Isabelle [41] or Coq [14]) in that KeY and its calculus are tailor-made for Java verification. The semantics of Java is “built in”. The program to be analyzed is kept as part of formulas. Constructing a proof in KeY corresponds to symbolic execution [31]. This helps to keep JavaDL formulas and proof goals human-readable and, thus, allows the user to understand the (sub-)proofs.

VI. THE CASE STUDY

In this section, we first provide a brief description of the Java program that we analyze: a simple e-voting system. We then state the security property that we verify and describe the verification process, which uses our hybrid approach to combine the strengths of the fully automatic tool Joana and the deductive verification system KeY. As already mentioned in the introduction, analysis of the e-voting system with Joana alone would have failed. However, using the hybrid approach, in conjunction with KeY, Joana can successfully be used to establish noninterference. KeY is required only to establish a specific functional property of the e-voting system. As already mentioned in the introduction, the analysis with KeY is almost finished, but ongoing work (see below).

A. Description of the Analyzed Program

The e-voting system involves voters, a server, and a bulletin board. Voters send their choices to the server using secure message transmission, i.e., using SMT\textsuperscript{Real}. The server, after having received the choices of all voters, computes the result of the election and posts it on the bulletin board in an authenticated way, i.e., it uses AMT\textsuperscript{Real} to send the result to the bulletin board. Everybody can now contact the bulletin board to learn the outcome of the election.

Besides (the code for) the voters, the server, and the bulletin board, the analyzed program also contains a setup class with the method main. This method models the interactions between the active environment (adversary), which controls the network and the components of the system in the following way.

First, the environment is asked to provide two vectors \(\vec{c}_0\) and \(\vec{c}_1\). These are the choices of the honest voters. The vectors contain \(n\) valid choices, one for each voter. The setup program checks that both vectors result in the same election outcome, i.e., the same number of (honest) votes for every candidate. If the two vectors do not fulfill this condition, the setup program aborts. Otherwise, one of these vectors, \(\vec{c}_b\), is picked, depending on the value of a secret bit \(b\). This secret bit (or more precisely, a static boolean variable \(b\)) is the only high variable in the program. The honest voters will vote according to \(\vec{c}_b\). For simplicity, here we assume all eligible voters to be honest (however, it should be possible to relax this assumption). Intuitively, our security requirement says that the environment does not learn \(b\), meaning that the environment does not learn whether the honest voters voted according to \(\vec{c}_0\) or \(\vec{c}_1\) (see also Section VI-B).

Next, the setup program creates objects representing eligible voters, the server, and the bulletin board. All those parties, including the environment, are registered to be able to use SMT\textsuperscript{Real} and/or AMT\textsuperscript{Real}.

Now, in a loop (which terminates when the environment says so) the environment decides which actions are executed. Those actions include triggering a chosen voter to vote, triggering the server to collect a vote, triggering the server to publish the result on the bulletin board, and reading the result from the bulletin board. In addition, the environment controls the network. Altogether, this models the capabilities an adversary/environment has in a distributed system.

We denote the described system, including the setup program, by \(E_{V,R}[b]\), where \(b\) is the boolean (high) variable mentioned above.

Part of the code of the e-voting system is provided in the appendix (see [11] for the full code and a runnable distributed system).

B. The Security Property

The security property we want to verify for the e-voting system is the (cryptographic) privacy of the votes of honest voters. Formally, this is expressed by a computational indistinguishability property (see Section II) as follows:

\[
E_{V,R}[\text{false}] \approx^0_{\text{comp}} E_{V,R}[\text{true}]. \tag{3}
\]

This property says that a polynomially bounded environment (adversary) is not able to distinguish whether the voting system \(E_{V,R}\) interacts with uses \texttt{true} or \texttt{false} as the initial value of the variable \(b\). By the definition of \(E_{V,R}\), this in turn means that even though the environment dictates the two possible ways in which the honest voters may vote (as long as they yield the same result), the environment is not able to tell which one of them has been actually used (except with negligible probability).

The computational indistinguishability relation in (3) uses the empty interface \(I = \emptyset\). This means that the environment cannot directly call methods defined in our e-voting system. However, the environment controls the network and by definition of \(E_{V,R}\) (in particular, the setup program) it can determine which actions are taken and when. We also note that \(E_{V,R}\) is an open system, which uses some classes not defined within \(E_{V,R}\), such as a network library. These classes
are provided by the environment,\textsuperscript{6} and hence, are untrusted. Thus, property (3) implies privacy of honest votes no matter how such untrusted libraries are implemented.

As briefly mentioned in Section VI-A, the server in $EV_{\bar{b}}$ is defined in such a way that it waits to receive all messages from the honest voters that vote according to $\bar{c}_0$ before it proceeds to publish the result of the election (possible requests of the environment to publish the result before all such votes have been received are ignored). We point out that this is important for the property (3) to hold. Otherwise, the environment could block the votes of all honest voters accept for one (for which $\bar{c}_0$ and $\bar{c}_1$ differ) and then, from the election outcome, it could easily deduce $b$ and, hence, whether the honest voters voted according to $\bar{c}_0$ or $\bar{c}_1$.

In order to prove (3), by Theorem 3 it suffices to show that

$$EV_{I[b]} \text{ is I-noninterferent,}$$

where $EV_{I}$ denotes the system which coincides with $EV_{R}$ except that the real protocols $SMT^{Real}$ and $AMT^{Real}$ are replaced by their ideal counterparts, $SMT^{Ideal}$ and $AMT^{Ideal}$, respectively.

Since, as can easily been seen, $EV_{I[b]}$ satisfies the conditions of Theorem 2, we can further reduce checking (4) to checking the following property:

$$E_{\bar{u}I_{\bar{u}}}EV_{I[b]} \text{ is noninterferent for all } \bar{u},$$

where the family of systems $E_{\bar{u}I_{\bar{u}}}$, parameterized by a finite sequence of integers $\bar{u}$, is as described in Section II. This system can be automatically generated from $EV_{I[b]}$. Also note that by “noninterference” we mean standard termination-insensitive noninterference (see Section II).

Altogether it suffices to prove (5) in order to obtain (3).

\textbf{C. Verification Process}

In principle, Joana is able to check property (5). However, when applied to check (5) for our particular program, Joana reports an information flow from the high value $b$ to the result of the election, and from this result to the low output.\textsuperscript{7} The reason for this alert is the over-approximation that Joana employs. The election result actually does not depend on $b$, because the setup $EV_{I[b]}$ ensures that the vectors $\bar{c}_0$ and $\bar{c}_1$ produce the same election result. If the server computes the result of the election correctly, then this result is independent of whether the honest voters voted according to $\bar{c}_0$ or to $\bar{c}_1$.

To avoid this false positive, an analysis tool has to establish that the server correctly computes the result of the election and that this result corresponds to $\bar{c}_0$ (and hence, $\bar{c}_1$). This is a non-trivial functional property, which — maybe not surprisingly — is beyond what Joana (and most or all other fully automated tools for checking non-interference) can achieve.

However, it is straightforward to provide an extension $EV_{I}^*$ of $EV_{I}$ which makes it more explicit for Joana that there is no information flow. If the program indeed works as expected, the election outcome corresponds to $\bar{c}_0$ (and hence, $\bar{c}_1$). So in the extension $EV_{I}^*$ we can explicitly state that the election outcome actually is the result corresponding to $\bar{c}_0$. More specifically, the result as determined by the vector $\bar{c}_0$, which in turn is determined by the environment, and hence, constitutes low data, is stored in an additional class $\text{CorrectResult}$ in $EV_{I}^*$ with static fields $\text{votesForA}$ and $\text{votesForB}$. (This class plays the role of $M$ in Definition 1): we note that this result is the same as the result determined by $\bar{c}_1$ (otherwise, the program is aborted). Second, after the point in the code of the server where the result of the election is computed by the server and stored in the local variables $\text{votesForA}$ and $\text{votesForB}$, we add the assignments

$$\text{votesForA} = \text{CorrectResult.votesForA};$$
$$\text{votesForB} = \text{CorrectResult.votesForB};$$

If the system is implemented correctly, then the result computed by the server indeed is the same as the correct result computed from the input and, therefore, the added assignments do not change the state of the program. In fact, we use the KeY tool to verify that in the state just before the additional assignments, the variables $\text{votesForA}$ and $\text{votesForB}$ already have the values $\text{CorrectResult.votesForA}$ and $\text{CorrectResult.votesForB}$, respectively (*). This then implies that $EV_{I}^*$ is a conservative extension of $EV_{I}$. (It is trivial to see that the additional class $\text{CorrectResult}$, which plays the role of $M$ in Definition 1, satisfies the required conditions.)

Proving (*) with KeY actually consists of two separate parts: (a) At the core is the proof that the computation of the election result is correctly implemented and, thus, corresponds to what is stored in $\text{CorrectResult}$. (b) We have to show that certain invariants hold that express well-formedness of data structures and the relation between the input and the actual data structures from which the result is computed.

The analysis with KeY of Part (a) is finished. This part involves 2 classes with 9 methods. These methods were annotated with about 160 lines of Java Modeling Language specification. The complete KeY proof for (a) consists of 11 sub-proofs with about 115,000 nodes in the proof trees (corresponding to rule applications). About 500 interactive steps were required to construct these proofs. KeY needs about one hour for the automated parts of proof construction on a standard desktop PC.
Although the proofs for Part (b) are less complex than the those for (a), there are more sub-proofs to consider. Hence, while straightforward, the analysis is more tedious. At the time of writing, this part of the proof is mostly finished but ongoing work.

The current status of the KeY proofs and the annotated programs are available [11]. This material will be updated once the analysis with KeY is finished.

Now, for the conservative extension $\mathcal{EV}_i^*$, Joana easily establishes the noninterference property

$$E_{\vec{i}'} \cdot \mathcal{EV}_i^*[b]$$

is noninterferent for all $\vec{i}'$. (6)

Joana took about 14 seconds on a standard PC (Core i5 2.3GHz, 8GB RAM) to finish the analysis of the program (with a size of 693 LoC). PDG computation took 12 seconds and only 2 seconds were needed to detect the absence of illegal flow inside the PDG (see also [11]). Note that the actual running code of the distributed system is much bigger than what Joana needed to analyze, because the code of the distributed system includes untrusted libraries, such as the standard Java library for networking, which do not need to be analyzed, as already mentioned in Section VI-B.

Therefore, once the analysis with KeY is completed, by Theorem 4, we conclude that (5) holds true. Thus, the cryptographic privacy property (3) follows as well.

REFERENCES


notions. As defined in [35], the form

\[ \langle e, s \rangle \]

is of the form \( \langle e, s \rangle \), where \( e \) is a Jinja+ expression and \( s \) is a state. A state is a pair \( \langle h, l \rangle \) of a heap and a store. A heap is a map from references (addresses) to object instances and a store is a map from variable names to values. A value can be either a reference or a value of a primitive type.

One particular type of expression is a block expression of the form \( \{ v : t ; e \}'C \) or \( \{ v : t ; v := \text{val} ; e \}'C \), where \( v \) is a local variable (whose scope is this block) of type \( t \) and, in the second variant, with value \( \text{val} \), \( e \) is an expression (\( e \) can access the local variable \( v \), and \( C \) is a class name (denoting that the block originates from the code of class \( C \)).

In general, an expression can contain many blocks as its subexpression. However, when we study expressions that occur in actual runs, it turns out that they have a simpler form, where all blocks are located on one path. Formally, let

\[ s_0 = \langle e_0, (h_0, l_0) \rangle \xrightarrow{t} \langle e_1, (h_1, l_1) \rangle \xrightarrow{t} \cdots \]

be a run with the initial state \( s_0 \). By the definition of the initial state [35], [36], \( h_0 \) is empty and \( l_0 \) bounds the static variables of the program to their initial values (and no other variables). By inspecting the rules of Jinja+ [36], one can see that, for every \( i = 0, 1, \ldots \),

- \( l_i \) bound only static variables,
- either \( e_i \) contains no block as its subexpression or \( e_i \) is of the form \( E[b] \), where \( E \) contains no block and \( b \) is a block. That is, \( e_i \) can contain, directly, at most one block (although \( b \) can contain other blocks).

The definitions given below will be applied to expressions originating from runs of Jinja+ systems and therefore they are defined only for expressions of the above form.

Pruning. Let \( C \) be a set of classes (intuitively, representing a subprogram) and \( e \) be an expression. We define a pruning operator \( \text{sub}_C(e) \) in such a way that it removes from \( e \) all those parts that come from classes not in \( C \) and only leaves the code originating in \( C \'. \) Formally, we define \( \text{sub}_C(e) \) as follows:

- if \( e \) contains no block, then \( \text{sub}_C(e) = e \),
- if \( e \) is not a block, but contains one, that is \( e = E[b] \), then \( \text{sub}_C(e) = E[\text{sub}(b)] \),
- if \( e = \{ v : t ; v := \text{val} ; e' \}'C \) with \( C \in C \)', then \( \text{sub}_C(e) = \{ v : t ; v := \text{val} ; \text{sub}(e') \}'C \), and similarly for \( e = \{ v : t ; e' \}'C \),
- if \( e = \{ v : t ; v := \text{val} ; e' \}'D \) with \( D \notin C \)', and \( e' \) contains no blocks, then \( \text{sub}_C(e) = \bot \),
- if \( e = \{ v : t ; v := \text{val} ; E(b) \}'D \) with \( D \notin C \)', then \( \text{sub}_C(e) = \text{sub}(b) \).

We will also use pruning operations for stores. Let \( C \) be a set of classes and \( I \) be an interface (that classes of \( C \) can use to access the rest of the program). Let \( I \) be a store. By \( \text{sub}_{C,I} \) we denote the store obtained from \( I \) by keeping only the static variable declared in \( C \) and \( I \) and removing all the other (static) variables.

Substates. The notion of a substate or a state of a component \( C \) in a program \( P \) can be captured by the following equivalence relation on states of programs. Let \( C \) be a set of classes (a program component) and \( I \) be an interface (that classes of \( C \) can use to access the rest of the program). For configurations \( c_0 \) and \( c_1 \), we write \( c_0 \sim_{C,I} c_1 \) if these configurations are the same up to (a) the code removed by \( \text{sub}_C \) and (b) up to a renaming of references (a bijection on the set of references).
Now, the substate of the system that is visible from a subsystem that defines classes in $C$ and uses an interface $I$ can be thought as the equivalence class of the relation $\sim_{C,I}$.

Now we define state equivalence formally. Let $f$ be a bijection on the set of references. We define a relation $\equiv_{l,f}$ on elements of the form $(v,h)$, where $v$ is a Jinja+ value and $h$ is a heap, as follows. For primitive values $v,v'$, we simply put $(v,h) \equiv_{l,f} (v',h')$ if and only if $v = v'$. For references $r,r'$, we have $(r,h) \equiv_{l,f} (r',h')$, if and only if the following conditions hold:

- $r' = f(r)$,
- the maximum class $D$ in the subtyping ordering, that (i) the actual class $l$ of the object pointed to by $r$ is a subclass of $D$ (in other words, $C$ extends $D$) and (ii) $D$ is in $C \cup I$ is the same as the corresponding maximal class of $r'$.

As one can see, two references (along with heaps) are in this relation, if the object structures rooted at them are the same, up to the given reference renaming, where one can only traverse those fields that are accessible for the given $C$ and $I$.

We can now define relation $\equiv_{l,f}$ on configurations, where $C$, $I$, and $f$ are as above. First, $(v,\langle h, I \rangle) \equiv_{l,f} (v',\langle h', I' \rangle)$, where $v$ and $v'$ are values, if

- $(v,h) \equiv_{l,f} (v',h')$ and
- the domains of $I$ and $I'$ are the same and, for each $x$ in the domain of $I$, we have $\langle l(x),h \rangle \equiv_{l,f} \langle l'(x),h' \rangle$.

Similarly we deal with references. Now, this relation can be extended to all configurations $(e,\langle h, I \rangle)$ by structural isomorphism on expressions $e$.

Now, we say that configurations $(e,\langle h, I \rangle)$ and $(e',\langle h', I' \rangle)$ are $(C,I)$-equivalent, written

$$(e,\langle h, I \rangle) \sim_{C,I} (e',\langle h', I' \rangle)$$

if there exists a bijection on references $f$ such that

$$\langle \text{sub}_{C}(e), \langle h, \text{sub}_{C,I}(l) \rangle \rangle \equiv_{l,f} \langle \text{sub}_{C}(e'), \langle h', \text{sub}_{C,I}(l') \rangle \rangle \rangle.$$

**Disjoint states.** Let $C$ and $I$ represents a program component, where $C$ is a set of classes and $I$ is an interfaced used by $C$ to access the rest of the program. Similarly, let and $C'$ and $I'$ represent a different component.

We say that, in a configuration $(e,\langle h, I \rangle)$, the components $(C,I)$ and $(C',I')$ have disjoint states, if the set of references occurring in sub$_{C}(e)$ and sub$_{C,I}(l)$ is disjoint with the set of references occurring in sub$_{C'}(e)$ and sub$_{C',I'}(l)$.

Two components $(C,I)$ and $(C',I')$ of a program $P$ have disjoint states, if they have disjoint states in all the configurations of the run of $P$.

If two components have disjoint states, then an action taken in one of the does not change the state of the second, unless the action is method call or return from a method:

**Lemma 3.** Let $c$ be a configurations in which $(C,I)$ and $(C',I')$ have disjoint states. Suppose that $c \xrightarrow{D} c'$ with $D \in C$ (that means that $c$ reduces to $c'$ in one step which is taken in the code of class $D \in C$), where this reduction is not obtained by the call rule [36, Rule (32)] nor the return rule [36, Rules (33) and (34)]. Then the state of $(C,I')$ is not changed by this step, i.e. $c \sim_{C,I'} c'$.

It is easy to show that, if only values of primitive types are exchanged between components, then the states of these components are separate:

**Lemma 4.** If the sets of static variables of $(C,I)$ and $(C',I')$ are disjoint and all the methods in $I$ and $I'$ are static methods with arguments of primitive types and returning values of primitive types, then $(C,I)$ and $(C',I')$ have disjoint states.

**Proof of Theorem 4.** With the above definitions and results, it is easy to prove Theorem 4. We first consider the simpler case, i.e., the notion of conservative extension as formalized in Definition 1. See below for the more general case.

By Lemma 4, class $M$ and the rest $R$ of the program $P'$ have disjoint states (the interface through which $M$ uses the rest of the program is empty, while the interface through which $P$ uses $M$ satisfies the assumptions of the lemma).

Let $\vec{y}$ be the vector of low variables which consists of all the variables of $P$ labeled as low. Recall that these variables are assumed to be static and of primitive types. Let $P[^{x}]$ be a conservative extension of $P[^{x}]$. Notice that neither the variables $\vec{x}$ nor $\vec{y}$ occur in $M$ (the additional class of $P'$).

Now, suppose that the program $P[^{x}]$ is noninterferent, whereas the original program $P[^{x}]$ is not. This means that there exist values $\vec{a}_1$ and $\vec{a}_2$ such that $P[^{x}]$ and $P[^{x}]$ terminate with final values $\vec{b}_1$ and $\vec{b}_2$, respectively, of the low variables $\vec{y}$ with $\vec{b}_1 \neq \vec{b}_2$.

Clearly, the variables $\vec{y}$ are static variables of the original program $P$. Therefore, one can show that, due to the stated above separation of $R$ and the state of $M$, and by Lemma 3, calls to $M$ do not change the state of $R$. Also, the additional assignments do not change the state of $R$, which trivially follows from the property of conservative extension. Hence, only action of $P'$ that change its state are the actions of the original program $P$.

Moreover, as it is assumed, calls to $M$ do not result in an uncaught exception.

Therefore, the programs $P[^{x}]$ and $P[^{x}]$ (1) terminate and (2) terminate with the same final state as, respectively.
by the assumption, does not change the state of the proof of Theorem 4, as the introduced new code, simply the proof). This extension does not require any changes in and PKISig (class SMT PKIEnc P not change the state of when the objects are cloned, but restored immediately after disjointness of states is temporarily violated, for the time straightforward way from the requirement that object are show that disjointness of states of changes: although we cannot use Lemma 4, it is easy to Theorem 4 remains true and its proof requires only minor requirement that methods of M first clone such objects and then operate only on these copies and do not use (nor store) the references obtained as parameters. For this extension, Theorem 4 remains true and its proof requires only minor changes: although we cannot use Lemma 4, it is easy to show that disjointness of states of M and R follows in a straightforward way from the requirement that object are cloned as the first carried out by methods of M (technically, disjointness of states is temporarily violated, for the time when the objects are cloned, but restored immediately after that).

Second, we can allow adding to P arbitrary code that does not change the state of P itself (which is denoted by R in the proof). This extension does not require any changes in the proof of Theorem 4, as the introduced new code, simply by the assumption, does not change the state of R.

Appendix B.
Functionality for Secure Message Transmission
A. Realization for SMT
The realization $SM^P_{\text{Real}}$ has the structure $SM \cdot PKI_{\text{Enc}}^P \cdot PKI_{\text{Sig}}^P$. Below we provide the full code of $SM$ (class $\text{Real.SMT}$). The code of the classes $PKI_{\text{Enc}}^P$ and $PKI_{\text{Sig}}^P$ is provided in [11].

```
// --- class Real.SMT ---
public class SMT {
    static public class SMTErr extends Exception {
    }
    static public class AuthenticatedMessage {
        public final byte[] message;
        public final int sender_id;
        private AuthenticatedMessage(byte[] message, int sender_id) {
            this.sender_id = sender_id;
            this.message = message;
        }
    }
    static public class AgentProxy {
        public final int ID;
        private PKI_{Enc}.Decryptor decryptor;
        private PKI_{Sig}.Signer signer;
        private AgentProxy(int ID, PKI_{Enc}.Decryptor decryptor,
                           PKI_{Sig}.Signer signer) {
            this.ID = ID;
            this.decryptor = decryptor;
            this.signer = signer;
        }
        public AuthenticatedMessage getMessage(int port) {
            try {
                byte[] inputMessage = NetworkServer.read(port);
                if (inputMessage == null) return null;
                // get the sender id and her verificer
                byte[] sender_id_as_bytes = MessageTools.first(inputMessage);
                int sender_id = MessageTools.byteArrayToInt(sender_id_as_bytes);
                PKI_{Sig}.Verifier sender_verifier = PKI_{Sig}.getVerifier(sender_id, 
                // retrieve the message with the recipient id
                byte[] message_with_recipient_id = MessageTools.second(inputMessage);
                int recipient_id = MessageTools.byteArrayToInt(message_with_recipient_id);
                message = encryptor.decrypt(signedAndEncrypted);
                if( !sender_verifier.verify(signature, message_with_recipient_id) )
                    return null; // invalid signature
                // make sure that the message is intended for this proxy
                byte[] message = MessageTools.second(message); 
                int recipient_id = MessageTools.byteArrayToInt(word.recipient_id); 
                if( recipient_id != ID ) 
                    return null; // message not intended for this proxy 
                message = MessageTools.second(message); 
                return new AuthenticatedMessage(message, recipient_id, inputMessage); 
            } catch (NetworkError | PKIError e) {
                return null;
            }
        }
    }
    public Channel channelTo(int recipient_id, 
                          String server, int port) 
        throws SMTErr, PKIError, NetworkError {
            if (registrationInProgress) throw new SMTErr();
            PKI_{Enc}.Encryptor recipient_encryptor = PKI_{Enc}.getEncryptor(recipient_id, 
            return new Channel(this.ID, recipient_id, this.sender_id, 
                               recipient_encryptor, server, port);
        }
    }
}
```
private Channel(int sender_id, int recipient_id, PKISig.Signer sender_signer, PKIEnc.Encryptor recipient_encryptor, String server, int port) {
    this.sender_id = sender_id;
    this.recipient_id = recipient_id;
    this.sender_signer = sender_signer;
    this.recipient_encryptor = recipient_encryptor;
    this.server = server;
    this.port = port;
}

public void send(byte[] message) {
    // sign and encrypt
    byte[] recipient_id_as_bytes = MessageTools.intToByteArray(recipient_id);
    byte[] message_with_recipient_id = MessageTools.concatenate(recipient_id_as_bytes, message);
    byte[] signed = sender_signer.sign(message_with_recipient_id);
    byte[] signedAndEncrypted = recipient_encryptor.encrypt(signed);
    byte[] outputMessage = MessageTools.concatenate(sender_id_as_bytes, signedAndEncrypted);
    try {
        NetworkClient.send(outputMessage, server, port);
    } catch (NetworkError e) {}
We will refer to this implementation of class SMT by Ideal.SMT.

C. Ideal Functionality PKIEnc\textsubscript{Ideal}

Because the simulator we use in the realization proof (presented in Section B-E) uses the ideal functionalities for PKI encryption and digital signatures, in this section we present the first of these functionalities, PKIEnc\textsubscript{Ideal}, with the second functionality presented in the next section.

```java
private static AgentProxy register(int id)
    throws SMTError, PKIError {
    if (registrationInProcess) throw new SMTError();
    registrationInProcess = true;
    // call the environment/simulator
    SMTEnv.register(id);
    // check whether the id has not been claimed
    if (registeredAgents.fetch(id) != null) {
        registrationInProcess = false;
        throw new PKIError();
    }
    AgentProxy agent = new AgentProxy(id);
    registeredAgents.add(agent);
    registrationInProcess = false;
    return null;
}

private static boolean registrationInProgress = false;
...
```

This implementation maintains a static collection of registered agents registeredAgents. Moreover, each decryptor (an object representing an agent) maintains a list log of messages encrypted by the functionality with a corresponding encryptor.

D. Ideal Functionalities PKISig\textsubscript{Ideal}

The ideal functionality PKISig\textsubscript{Ideal} is implemented as follows (we omit some parts of the program, in particular the code that implements digital signature primitives; the full code is available in [11]).

```java
private byte[] publicKey;
private byte[] privateKey;
private EncryptionLog log;
...
```

```java
public static Decryptor register(int id) throws PKIError {
    if (Environment.untrustedOutput(id) == 0)
        throw new NetworkError();
    return new Decryptor(id);
}
```

```java
public static Decryptor getEncryptor(int id)
    throws NetworkError, PKIError {
    if (Environment.untrustedInput() == 0)
        throw new NetworkError();
    Decryptor enc = registeredAgents.fetch(id);
    if (enc == null)
        throw new PKIError();
    return enc;
    ...
```

```java
public static Decryptor getEncryptor(int id) throws PKIError {
    Decryptor decryptor = new Decryptor(id);
    return decryptor;
}
```

```java
public Encryptor encrypt(byte[] message) {
    byte[] messageCopy = copyOf(message);
    if (!log.containsCiphertext(messageCopy)) {
        return copyOf(CryptoLib.pke_decrypt(
            copyOf(privateKey), messageCopy));
    } else {
        return copyOf(log.lookup(messageCopy));
    }
}
```

```java
public Encryptor getEncryptor() {
    return new Encryptor(ID, publicKey, log);
}
```
private Verifier(int id, byte[] verifKey, Log log) {
    this.ID = id;
    this.verifKey = verifKey;
    this.log = log;
}

public boolean verify(byte[] signature, byte[] message) {
    if (!CryptoLib.verify(message, signature, verifKey))
        return false;
    return log.contains(message);
}

public byte[] getVerifKey() {
    return copyOf(verifKey);
}

public class Signer {
    private int ID;
    private byte[] verifKey;
    private Log log;

    private Signer(int id) {
        KeyPair keypair = CryptoLib.generateSignatureKeyPair();
        this.signKey = copyOf(keypair.publicKey);
        this.log = new Log();
    }

    public byte[] sign(byte[] message) {
        byte[] signature = CryptoLib.sign(copyOf(message),
                                            copyOf(signKey));
        if (signature == null) return null;
        if (!CryptoLib.verify(copyOf(message),
                              copyOf(signature),
                              copyOf(verifKey)))
            return null;
        log.add(copyOf(message));
        return copyOf(copyOf(signature));
    }

    public Verifier getVerifier() {
        return new Verifier(ID, verifKey, log);
    }

    public static Signer register(int id) {
        if (registeredAgents.fetch(id) != null) return null;
        Signer signer = new Signer(id);
        Verifier verifier = signer.getVerifier();
        registeredAgents.add(verifier);
        return signer;
    }

    public static Verifier getVerifier(int id) {
        return registeredAgents.fetch(id);
    }
}

As in the case of PKIEncIdeal, this implementation maintains a static collection of registered agents registeredAgents. Moreover, each Signer object (an object representing an agent) maintains a list log of signed messages. Whenever a message is to be verified using a corresponding Verifier object, it is made sure that this message is in log.

E. The Simulator

In this section we present the simulator that will be used in the next section. In this presentation we leave out the (straightforward) implementation of collections (the full code is available in [11]).

```java
// --- class SMTSimulator.SMTEnv -- //
public class SMTEnv {
    public static void register(int id) {
        try {
            AgentProxy proxy = SMT.register(id);
            agentProxies.add(proxy);
        } catch (PKIError | SMT.SMTError e) {} }

    public static boolean channelTo(int sender_id, int recipient_id, String server, int port) {
        AgentProxy sender = agentProxies.fetch(sender_id);
        try {
            Channel channel = sender.channelTo(recipient_id, server, port);
            channels.add(channel, sender_id, recipient_id, server, port);
        } catch (PKIError | SMT.SMTError e) {} catch (NetworkError e) {
            return false;
        }
        return true;
    }

    public static byte[] send(int message_length, int sender_id, int recipient_id, String server, int port) {
        Channel channel = channels.fetch(sender_id, recipient_id, server, port);
        byte[] message = MessageTools.getZeroMessage(message_length);
        byte[] output_message = channel.send(message);
        agentProxies.getMessageQueue(recipient_id).add(output_message);
        return output_message;
    }

    public static int getMessage(int id, int port) {
        try {
            AgentProxy proxy = agentProxies.fetch(id);
            byte[] input_message = proxy.getMessage(port);
            if (input_message == null) return -1;
            int index = agentProxies.getMessageQueue(id).getIndex(input_message);
            return index;
        } catch (SMT.SMTError e) {
            return -1;
        }
    }
}
```

In this program, SMT denotes a (slightly changed) copy of Real.SMT. This copy of Real.SMT will be denoted by SMTSimulator.SMT below. The changes made in this copy of Real.SMT are:

- method Channel.send returns outputMessage, instead of sending this message to the network,
method agentProxies.getMessage returns inputMessage (instead of an authenticated message).

The simulator $\text{SMTSim}$ is the system

$$\text{SMTSim} = \text{SMTSimulator.SMTEnv} \cdot \text{SMTSimulator.SMT} \cdot \text{PKIEnc}_{\text{Ideal}} \cdot \text{PKISig}_{\text{Ideal}}.$$ 

Note that $\text{SMTSimulator.SMT} \cdot \text{PKIEnc}_{\text{Ideal}} \cdot \text{PKISig}_{\text{Ideal}}$ is a (slightly modified version of) $\text{SMT}^*$. 

Object agentProxies is a collection of objects of class AgentProxy defined in $\text{SMTSimulator.SMT}$. The simulator uses this collection to store the proxies created, and returned by $\text{SMTSimulator.SMT}$. Moreover, when a message is sent to some recipient, the output message returned by $\text{SMTSimulator.SMT}.\text{Channel}.\text{send}$ is added to the message queue of the recipient. This is used by the simulator to return an index into the list used by the method getMessage of the ideal SMT.

Similarly, channels is a collection of channels (plus some additional data used for retrieval) created and returned by the simulated functionality $\text{SMTSimulator.SMT}$. 

F. From the Real Functionality to the Intermediate One

As we have explained, the real functionality for SMT, $\text{SMT}_{\text{Real}}$, consists of class $\text{Real.SMT}$ that uses $\text{PKIEnc}_{\text{Real}}$ and $\text{PKISig}_{\text{Real}}$ as its building blocks. Thus,

$$\text{SMT}_{\text{Real}} = \text{Real.SMT} \cdot \text{PKIEnc}_{\text{Real}} \cdot \text{PKISig}_{\text{Real}}.$$ 

Our realization proof is modular and uses properties of simulatability proven in [35], namely transitivity of simulatability and the composition theorem (also discussed in Section II).

We will use the fact that $\text{PKIEnc}_{\text{Real}}$ realizes $\text{PKIEnc}_{\text{Ideal}}$ w.r.t. the interfaces $\text{IPKIEnc}$, $\emptyset$, $\text{INet}$, $\text{ISimPKIEnc}$ and that $\text{PKISig}_{\text{Ideal}}$ realizes $\text{PKISig}_{\text{Ideal}}$ w.r.t. the interfaces $\text{IPKISig}$, $\emptyset$, $\text{INet}$, $\text{ISimPKISig}$. 

By applying the composition theorem [35], we thus conclude that $\text{PKIEnc}_{\text{Real}}, \text{PKISig}_{\text{Real}}$ realizes $\text{PKIEnc}_{\text{Ideal}}, \text{PKISig}_{\text{Ideal}}$ w.r.t. $\text{IPKIEnc} \cup \text{IPKISig}$, $\emptyset$, $\text{INet}$, $\text{ISimPKIEnc} \cup \text{ISimPKISig}$. Now, it follows from the definition of simulatability that $\text{PKIEnc}_{\text{Real}}, \text{PKISig}_{\text{Real}}$ realizes $\text{PKIEnc}_{\text{Ideal}}, \text{PKISig}_{\text{Ideal}}$ also w.r.t. $\text{IPKIEnc} \cup \text{IPKISig}, \text{ISMT}, \text{INet}, \text{ISimPKIEnc} \cup \text{ISimPKISig}$. 

By reflexivity of simulatability [35, Lemma 1], $\text{SMT}$ realized $\text{SMT}$ (that is, itself) with respect to $\text{ISMT}, \text{IPKIEnc} \cup \text{IPKISig}, \text{INet}, \emptyset$. It now follows from the composition theorem that $\text{SMT} \cdot \text{PKIEnc}_{\text{Real}} \cdot \text{PKISig}_{\text{Real}}$ realizes $\text{SMT} \cdot \text{PKIEnc}_{\text{Ideal}} \cdot \text{PKISig}_{\text{Ideal}}$ w.r.t. $\text{ISMT} \cup \text{IPKIEnc} \cup \text{IPKISig}$, $\emptyset$, $\text{INet}$ $\text{ISimPKIEnc} \cup \text{ISimPKISig}$. 

Let $\text{SMT}_{\text{Real}}$ denote $\text{Real.SMT} \cdot \text{PKIEnc}_{\text{Ideal}} \cdot \text{PKISig}_{\text{Ideal}}$. By the definition of simulatability, as an immediate consequence of the above fact, we obtain the following result.

Lemma 5. $\text{SMT}_{\text{Real}}$ realizes $\text{SMT}_{\text{Real}}$ w.r.t. the interfaces $\text{ISMT}, \emptyset, \text{INet}, (\text{ISimPKIEnc} \cup \text{ISimPKISig})$. 

G. From the Intermediate Functionality to the Ideal One 

Int this section we prove the following fact:

Lemma 6. $\text{SMT}^*$ realizes $\text{SMT}_{\text{Ideal}}$ w.r.t. the interfaces $\text{ISMT}, \emptyset, (\text{INet} \cup \text{ISimPKIEnc} \cup \text{ISimPKISig}), \text{ISimSMT}$. 

In this proof we use the simulator $\text{SMTSim}$ defined in Section B-E. We want to prove that

$$\text{SMT}^* = \text{Real.SMT} \cdot \text{PKIEnc} \cdot \text{PKISig}$$ 

and

$$\text{SMT}_{\text{Ideal}} \cdot \text{SMTSim} = \text{Ideal.SMT} \cdot \text{SMTSimulator.SMTEnv} \cdot \text{SMTSimulator.SMT} \cdot \text{Ideal.PKIEnc} \cdot \text{Ideal.PKISig}.$$ 

are computationally indistinguishable w.r.t. $\text{ISMT}$. 

We denote by $\bar{S}$ the system $\text{SMT}^*$, and by $\bar{S}$ the system $\text{SMT}_{\text{Ideal}} \cdot \text{SMTSim}$. By $q, q', q''$ we will denote configurations of $S \cdot E$ and by $q, q', q''$ configurations of $\bar{S} \cdot E$, where $E$ is an environment.

The general idea of the proof is the following. First, we establish a strict relation between the state of $\text{Ideal.SMT}$ and the state of the simulator including the simulated functionalities $\text{SMTSimulator.SMT}, \text{Ideal.PKIEnc}$, and $\text{Ideal.PKISig}$ in the configurations of the run of $\bar{S} \cdot E$. This relation, formalized by the notion of consistent states, includes, for instance, the requirements that the same agents are registered in both components. Second, we establish a relation between the run of $S$ and the run of $\bar{S}$, where, in particular, in corresponding configurations which are controlled by the environment, the state of $E$ is the same.

These relations are preserved by steps made by $E$ (which is a consequence of the separation result stated below) and, as one can show by analysing the corresponding calls to public method of $S$ and $\bar{S}$, are also preserved by those methods.

Separation of states. In the results stated below we use the notions introduced in Appendix A.

Although, in the run of $S \cdot E$ and $\bar{S} \cdot E$, the state of the subsystem $S/S$ is not completely separated from the state of $E$, we can show nearly complete separation, where only references of classes AgentProxy and Channel can be shared by both subcomponents of the system. Because objects of these classes cannot be directly modified nor inspected by the code of the environment (except for reading the read-only field ID of AgentProxy), this still provides high level of separation that can be leveraged in the proof.

Formally, we state this separation as follows. Let $I_E$ be the interface provided by the environment such that $I_E \vdash S$ and $I_E \vdash \bar{S}$. Let $I_S = I_S = ISMT$. In the following, we use $E$ for the set of classes defined in $E$ and similarly for $S$ and
\( \tilde{S} \). The following facts follow from the observation that only objects of classes \texttt{AgentProxy} and \texttt{Channel} are exchanged by the method calls between the subcomponents.

**Lemma 7.** Let \((e, (l, h))\) be a configuration in the run of the system \( \tilde{S} \cdot E \), for some environment \( E \). Then the set of references occurring in \( \text{sub}_S(e) \) and \( \text{sub}_{\tilde{S}_E}(l) \) is disjoint with the set of references occurring in \( \text{sub}_E(e) \) and \( \text{sub}_{\tilde{E}_E} \), except for references to objects of classes \texttt{Real.SMT.AgentProxy} and \texttt{Real.SMT.Channel}.

**Lemma 8.** Let \((e, (l, h))\) be a configuration in the run of the system \( \tilde{S} \cdot E \), for some environment \( E \). Then the set of references occurring in \( \text{sub}_S(e) \) and \( \text{sub}_{\tilde{S}_E}(l) \) is disjoint with the set of references occurring in \( \text{sub}_E(e) \) and \( \text{sub}_{\tilde{E}_E} \), except for references to objects of classes \texttt{Ideal.SMT.AgentProxy} and \texttt{Ideal.SMT.Channel}.

One can prove that this level of separation is enough to guarantee the following fact, which is analogous to Lemma 3 and, informally, says that the actions taken by \( E \) do not change the state of \( S/\tilde{S} \) and, conversely, that the actions take by \( S/\tilde{S} \) do not change the sate of \( E \).

**Lemma 9.** Let \( q \) be a configuration in the run of the system \( S \cdot E \), for some environment \( E \).

1. Suppose that \( q \xrightarrow{D} q' \) with \( D \in E \) (i.e. this step is executed by \( E \)), where this reduction is not obtained by the call rule [36, Rule (32)] nor the return rule [36, Rules (33) and (34)]. Then \( q \sim_{\tilde{S}_E} q' \).

2. Suppose that \( q \xrightarrow{D} q' \) with \( D \in S \), where this reduction is not obtained by the call rule nor the return rule. Then \( q \sim_{E} q' \).

**Lemma 10.** Let \( \tilde{q} \) be a configuration in the run of the system \( \tilde{S} \cdot E \), for some environment \( E \).

1. Suppose that \( \tilde{q} \xrightarrow{D} \tilde{q}' \) with \( D \in E \), where this reduction is not obtained by the call rule nor the return rule. Then \( \tilde{q} \sim_{\tilde{S}_E} \tilde{q}' \).

2. Suppose that \( \tilde{q} \xrightarrow{D} \tilde{q}' \) with \( D \in \tilde{S} \), where this reduction is not obtained by the call rule nor the return rule. Then \( \tilde{q} \sim_{E} \tilde{q}' \).

**Recursion depth.** Let \((e, (h, l))\) be a configuration. The recursion depth of \( c \) is the number of nested blocks in \( e \) (as we have noted in Section A, there is only one path in \( e \) on which block are located).

Note that if \( c \xrightarrow{\ell} c' \) by the call rule, then the recursion depth of \( c' \) is bigger than the recursion depth of \( c \). Conversely, if \( c' \) is obtained by a return rule, its recursion depth decreases.

A block of level \( k \) is a sequence of configurations \( c_1, \ldots, c_n \) such that

\[ c_1 \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_{n-1}} c_n \]

where the recursion depth of \( c_1 \) and \( c_n \) is \( k \), and the recursion depth of \( c_2, \ldots, c_{n-1} \) is at least \( k \). Intuitively, a block of level \( k \) is a part of run obtained by executing the code of some method, including sub-calls (but without returning from such a method). Such a block is a \emph{finished}, if the last configuration reduces (in one step) using a return rule.

We say that a configuration \( q \) is a \( E \)-configuration, if \( E \) has control at \( c \), i.e. \( c \xrightarrow{E} c' \) for \( c' \in E \).

A (finished) \( E \)-block of level \( k \), is a (finished) block of level \( k \), as above, such that the first and the last configuration are \( E \)-configurations. Similarly, we define a (finished) \( S \)-block and \( \tilde{S} \)-block.

**Consistent configurations.** Now, we introduce a notion of a consistent configuration which is a configuration, when different parts of the system \( S \) are synchronized (as for who is registered) and, moreover, the simulator keeps hold on all the crucial object created by class \texttt{SMTSimulator} \cdot \texttt{SMT} is uses.

Formally, we say that a configuration \( \tilde{q} \) is \emph{consistent}, if the following conditions hold:

(a) If the static field \texttt{registrationInProgress} of class \texttt{Ideal.SMT} is set to false, then the following three statements are equivalence: (1) an identifier \texttt{id} is registered in \texttt{Ideal.SMT.registeredAgents}, (2) \texttt{id} is registered in \texttt{PKIEnc.registeredAgents}, and (3) \texttt{id} is registered in \texttt{PKISig.registeredAgents}.

(b) All the objects of class \texttt{SMTSimulator} \cdot \texttt{SMT.AgentProxy} created so far are stored in \texttt{SMTEnv.agentProxies}. There is at most one such object for each agent \texttt{id}.

(c) Each objects of class \texttt{SMTSimulator} \cdot \texttt{SMT.Channel} created so far is stored in \texttt{SMTEnv.channels} along with the same data (sender identifier, recipient identifier, sever, and port) as was used to create the channel.

(d) The length of the list queue of an object of class \texttt{Ideal.SMT.AgentProxy} with \texttt{ID} = \texttt{id} is the same as the length of the list of messages for the agent with identifier \texttt{id} maintained by the simulator (class \texttt{SMTEnv}) (this list is accessible through \texttt{SMTEnv.agentProxies.getMessage}). Moreover, the latter list contains distinct elements.

**Lemma 11.** Every \( E \)-configuration in the run of \( \tilde{S} \cdot E \) is consistent.

**Proof sketch:** Let \( K \) be the maximal recursion depth of a configuration in the run \( \rho \) of \( \tilde{S} \cdot E \).

We will prove the following statement for all \( k \). If \( B \) is a finished block of level \( k \) such that the initial configuration of \( B \) is consistent, then all the \( E \)-configurations in \( B \) are consistent. Moreover, if this block is an \( \tilde{S} \) block that was obtained by calling a public method of \( \tilde{S} \), then the final state is also consistent.

This is enough to complete the proof of the lemma. Indeed, the complete run \( \rho \) is a finished \( E \)-block of level 1
We prove the above statement by induction on \( k \): we prove it for a given \( k \), assuming that it holds for all \( k' > k, k' \leq K \).

First, let us consider the case when \( B \) is an \( E \)-block. By Lemma 10, steps executed within the code of \( E \) preserve the state of \( \tilde{S} \) and thus preserve consistency (which is a property of the state of \( S \)). So, whenever \( E \) calls \( \tilde{S} \), it does it in a consistent state and the run enters a block with \( k' > k \). Therefore, we can use the inductive hypothesis to conclude that (1) all the \( E \)-configuration within this block are consistent and (2) after such a call \( E \) continues in a consistent state. Therefore, consistency is preserved to the end of the block.

Now, let us consider the case when \( B \) is an \( \tilde{S} \) block. It is enough to consider all block obtained when a public method of \( \tilde{S} \) is called in a consistent state. We proceed on the case by case basis.

To illustrate the details of the proof, let us consider here method \( \text{Ideal.SMT.Channel.send} \) called for an object \( ch \) with the argument \( m \). This method does not change the set of registered agents nor the set of created objects of \( \text{class agentProxies} \) or \( \text{Channel} \). Therefore, the execution of this method does not affect conditions (a)-(c) of the definition of consistent configurations. The lists considered in condition (d), however, are changed by this method. Nonetheless, it is easy to show that this condition is not violated when this methods gives control to the environment, which happens (1) when \( \text{output_message} \) is sent by the method \( \text{Ideal.SMT.Channel.send} \) and (2) when this method returns. This is because the length of both list has been increased by one and a fresh message has been added to the latter list (as the ideal functionality for encryption guarantees that ciphertexts for a given party are distinct). Note that the simulator operates on the corresponding channel to the one that is used by the ideal functionality for SMT, because of the assumed consistency of initial states.

Note that when (1) takes place, the environment is called, which means that a new block is created of a higher level than \( k \). As the program execution enters this block in a consistent state (what we just have shown), by the induction hypothesis, all the \( E \)-states in this block are consistent.

**Corresponding states.** Let \( T \cdot S \) and \( T' \cdot S' \) be programs, and \( I \) be the interface that \( I \vdash T \). We say that the state of \( T \) in a configuration \( c = (e, (h, l)) \) of \( T \cdot S \) and the state of \( T \) in a configuration \( c' = (e', (h', l')) \) of \( T' \cdot S' \) are the same, if there exists a bijection \( f \) on the set of references such that:

\[
\langle \text{sub}_T(e), \langle h, \text{sub}_{T,I}(l) \rangle \rangle \equiv_{T,I,f} \langle \text{sub}_{T'}(e'), \langle h', \text{sub}_{T,I}(l') \rangle \rangle.
\]

Let \( q \) be a state of \( S \cdot E \) and \( \bar{q} \) be a state of \( \tilde{S} \cdot E \), for some environment \( E \). We say that \( q \) and \( \bar{q} \) are corresponding, if:

(a) The state of \( E \) in \( q \) is the same as the state of \( E \) in \( \bar{q} \).

(b) The following statements are equivalent:

- The collection \( \text{queue} \) of an object of class \( \text{Ideal.SMT.agentProxies} \) with \( \text{ID} = \text{rid} \) contains a pair \( (m, \text{sid}) \) at position \( i \).

- A unique message \( c \) is stored in the list of messages for the agent with identifier \( \text{rid} \) maintained by the simulator (\( \text{class SMTSimulator.SMTEnv} \)) at the same position \( i \) such that \( c \), when processed by the code in the method \( \text{Real.SMT.AgentProxy.getMessage} \) for the agent with identifier \( \text{rid} \) and with \( \text{inputMessage} = c \), results in an authenticated message \( (m, \text{sid}) \).

Let \( q \) be an \( E \)-configuration. We write \( q \mapsto q' \), if \( q' \) is also an \( E \)-configuration and

\[
q \mapsto q_0 \mapsto q_1 \mapsto \cdots \mapsto q_n \mapsto q' \]

where \( q_0 \in E \) and \( q_i \not\in E \) for \( i \in \{1, \ldots, n\} \), that is \( q_1, \ldots, q_n \) are not \( E \)-configurations. (Note that the special case of the above rule is when \( q' \) is obtained from \( q \) in one step). As we can see, \( q' \) is the next \( E \)-configuration after \( q \).

Now, using Lemma 11, we can prove the following fact.

**Lemma 12.** Let \( q_0 \) be the initial configuration of the run of \( S \cdot E \) and \( \bar{q}_0 \) be the initial configuration of the run of \( \tilde{S} \cdot E \). Let \( q_1, \ldots, q_n \) and \( \bar{q}_1, \ldots, \bar{q}_n \) be configurations such that \( q_0 \mapsto q_1 \mapsto \cdots \mapsto q_n \) and \( \bar{q}_0 \mapsto \bar{q}_1 \mapsto \cdots \mapsto \bar{q}_n \) where \( q_n \) and \( \bar{q}_n \) are final configuration (i.e. configuration that do not reduce).

Then \( n = m \) and, for all \( i \in \{0, \ldots, n\} \), the configurations \( q_i \) and \( \bar{q}_i \) are corresponding.

**Proof sketch:** Let us observe that, if \( q \) and \( \bar{q} \) are corresponding, then the number of \( E \)-block in those states are the same.

We will prove a more general fact, than the one stated in the lemma, allowing \( q_0 \) and \( \bar{q}_0 \) to be any corresponding \( E \)-states. The proof proceeds by induction on the number of \( E \)-blocks in \( q_0 \) (and \( \bar{q}_0 \)), where to prove that the statement is true for configurations with a given number of \( E \)-blocks, we assume that it holds true for configurations with bigger numbers of \( E \)-blocks.

If \( q \) and \( \bar{q} \) are corresponding and \( q \mapsto q' \) where \( q' \) is obtained from \( q \) in one step, then \( \bar{q} \mapsto \bar{q}' \) where \( \bar{q}' \) is obtained also in one step. Moreover, one can show that, in such a case, \( q' \) and \( \bar{q}' \) are corresponding (as these steps are \( E \) steps, the state of \( S/\tilde{S} \) is not changed, due to the proven state separation, and the state of \( E \), which is the same in both cases, change in the same way). It means that the actions of \( E \) preserve the correspondence of states. Therefore, it is enough to show that (public) method of \( S \) and \( \tilde{S} \) do not break this property. More precisely, we consider, on the case by case basis, all calls from \( E \) to public methods of \( S \) and \( \tilde{S} \) respectively, made in corresponding states and show that they end in corresponding states.
Let us consider, for instance, method `Channel.send` called for an object `ch` with sender `sid` and recipient `rid` and with the argument `m` called in corresponding states of `S` and `S̃`.

First, we can observe, that by the separation of states the code of `S` and `S̃` do not change the state of `E`. Therefore it is enough to show that such a call preserves condition (b) of the definition of consistent states.

When this method is invoked, in both system, the interaction with the environment is the same: first, the output network method is called with some message and, second, the control returns back to the environment. Therefore, by the inductive hypothesis, it is enough to show that the networks are in corresponding states when those events occur.

As in both systems sending the message over the network is the last step, the following reasoning applies to both of them.

In the real system `S`, a message `c` is sent in the last step of the method. This message is obtained from `m` by signing and encrypting in the code `Real.SMT.Channel.send`. Note that `c`, when processed by the proxy of the agent with identifier `rid`, results in `(m,sid)`.

On the other hand side, in the ideal system `S̃`, the following happens:

- The same message `c` is computed (this is due to the fact that we use here ideal functionality of encryption which produces a cipher-text independent of the plaintext being encrypted). This step is executed by the simulated functionality SMT.
- `c` is stored in the list of messages for the agent with identifier `rid` maintained by the simulator at same position `i` (end of the list). This step is made by the code of `SMTSimulator.SMTEnv`.
- The fact that `c` is stored in the list of the same agent `rid` follows from Lemma 11 which guarantees that the states are consistent, and therefore the simulator uses the channel object corresponding to the channel object used by `Ideal.SMT`.
- Then `(m,sid)` is added to the collection queue of an object of class `Ideal.SMT.agentProxies` with ID = `rid` at the position with the same index `i`.
- `c` is sent over the network.

As we can see, in both systems, the same message `c` is sent over the network and when it happens (and right after this, when the method returns) condition (b) of the definition of corresponding states is satisfied (by the above observations).

Now we can complete the proof of Lemma 6. By the above lemma, the final configurations of `S · E` and `S̃ · E` (that is, respectively, `q_n` and `q̃_n`) are corresponding, which means, in particular, that the state of `E`, which includes the variable result in those configurations is the same. Therefore the environment outputs the same result in both cases.  

H. Proof of Theorem 5

Now, the main result of this section (Theorem 5) immediately follows from Lemma 1 and Lemma 2. We restate this result, providing more details on the involved interfaces:

**Theorem 7.** `SMTReal realizes SMTIdeal w.r.t. the interfaces `I_{SMT}, ∅, I_Neo, (I_{SimSMT} ∪ I_{SimPKISig})`.  

APPENDIX C.

THE CODE FOR THE CASE STUDY

Below we present part of the code of the case study (see [11] for the full code).

A. The Voter

```java
public class Voter {
    private final byte vote;
    private final SMT.Channel channel.to.server;

    public Voter(byte vote, SMT.AgentProxy voter.proxy)
        throws SMTError, PKIError, NetworkError {
        this.vote = vote;
        this.channel.to.server
            = voter.proxy.channelTo(
                Identifiers.SERVER.ID,
                Parameters.DEFAULT_LISTEN_PORT_SERVER_SMT);
    }

    public void onSendBallot() throws SMTError {
        byte[] ballot = new byte[] {vote};
        channel.to.server.send(ballot);
    }
}
```

B. The Server

```java
public class Server {
    public static final int NumberOfVoters = 50;
    private final boolean[] ballotCast;
    private int votesForA;
    private int votesForB;
    private final SMT.AgentProxy samt_proxy;
    private final AMT.Channel channel.to.BB;

    public Server(SMT.AgentProxy samt.proxy, AMT.AgentProxy amt.proxy)
        throws AMTError, PKIError, NetworkError {
        this.samt_proxy = samt.proxy;
        votesForA = 0;
        votesForB = 0;
        channel.to.BB
            = amt_proxy.channelTo(
                Identifiers.BULLETIN.BOARD.ID,
                Parameters.DEFAULT_LISTEN_PORT_BBBOARD_SMT);
        ballotCast = new boolean[NumberOfVoters];
    }
}
```

```java
/* Collect one ballot (read from a secure channel) */

public void onCollectBallot() throws SMTError {
    SMT.AuthenticatedMessage am = samt_proxy.getMessage(
        Parameters.DEFAULT_LISTEN_PORT_SERVER_SMT);
    ```
```
C. The Bulletin Board

```java
public class BulletinBoard {
    public BulletinBoard(AgentProxy proxy) {
        content = new MessageList();
        am_proxy = proxy;
    }

    public void onPost() throws AMTError {
        if (am == null) return;
        int voterID = am.sender_id;
        byte[] ballot = am.message;

        if (voterID == 0 || voterID > NumberOfVoters) return;
        int ballotCast[voterID] return;
        byte[] ballot = [voterID] = true;
        if (ballotCast[voterID] == true) {
            if (ballotCast[0] || ballot.length == 1) return;
            int candidate = ballot[0];
            if (candidate == 0) ++votesForA;
            if (candidate == 1) ++votesForB;
        }

        public boolean resultReady() {
            for (int i = 0; i < NumberOfVoters; ++i) {
                if (ballotCast[i] == true) {
                    return false;
                }
            }
            return true;
        }

        public void onSendResult(String addr, int port) throws NetworkError {
            byte[] result = getResult();
            if (result != null) {
                NetworkClient.send(result, addr, port);
            }
        }

        public void onPostResult() throws AMTError {
            byte[] result = getResult();
            if (result != null) {
                channel_to_BB.send(result);
            }
        }

        private byte[] getResult() {
            if (!resultReady()) return null;
        }

        private static byte[] formatResult(int a, int b) {
            String s = "Result of the election:";
            if (votesForA == 0) votesForA = HonestVotersSetup.CorrectResult.votesForA;
            if (votesForB == 0) votesForB = HonestVotersSetup.CorrectResult.votesForB;
            return formatResult(votesForA, votesForB);
        }
    }
}
```

D. The Setup

```java
public class HonestVotersSetup {
    static class Adversary {
        public final SMTChannel channel_to_server;
        public final AMTChannel channel_to_BB;
    }

    public void onAdversary() throws SMTError, PKIError, AMTError, NetworkError, AMTError {
        SMTChannel channel_to_server = SMT.register(Identifiers.ADVERSARY_ID);
        channel_to_BB = adversary_samt_proxy.channelTo(Identifiers.SERVER_ID, "www.server.com", 89);
        AMTChannel channel_to_BB = adversary_amt_proxy.channelTo(Identifiers.BULLETIN_BOARD_ID, "bulletinboard.com", 89);
    }
}
```

```java
class MessageList {
    class Node {
        byte contentMessage, node.message;
    }

    private MessageList content;
    private AMT.AgentProxy am_proxy;
}
```

```
D. The Setup

```java
public class HonestVotersSetup {
    static class Adversary {
        public final SMTChannel channel_to_server;
        public final AMTChannel channel_to_BB;

        public void onAdversary() throws SMTError, PKIError, AMTError, NetworkError, AMTError {
            SMTChannel channel_to_server = SMT.register(Identifiers.ADVERSARY_ID);
            channel_to_BB = adversary_samt_proxy.channelTo(Identifiers.SERVER_ID, "www.server.com", 89);
            AMTChannel channel_to_BB = adversary_amt_proxy.channelTo(Identifiers.BULLETIN_BOARD_ID, "bulletinboard.com", 89);
        }
    }

    static class CorrectResult {
        static public int votesForA = 0;
        static public int votesForB = 0;
    }
```

```java
D. The Setup

```java
public class HonestVotersSetup {
    static class Adversary {
        public final SMTChannel channel_to_server;
        public final AMTChannel channel_to_BB;

        public void onAdversary() throws SMTError, PKIError, AMTError, NetworkError, AMTError {
            SMTChannel channel_to_server = SMT.register(Identifiers.ADVERSARY_ID);
            channel_to_BB = adversary_samt_proxy.channelTo(Identifiers.SERVER_ID, "www.server.com", 89);
            AMTChannel channel_to_BB = adversary_amt_proxy.channelTo(Identifiers.BULLETIN_BOARD_ID, "bulletinboard.com", 89);
        }
    }

    static class CorrectResult {
        static public int votesForA = 0;
        static public int votesForB = 0;
    }
```
private static byte[] chooseVoterChoices(
    byte[] voterChoices1, byte[] voterChoices2) {
  byte[] voterChoices = new byte[Server.NumberOfVoters];
  for (int i=0; i<Server.NumberOfVoters; ++i) {
    final byte data1 = voterChoices1[i];
    final byte data2 = voterChoices2[i];
    voterChoices[i] = (secret ? data1 : data2);
  }
  return voterChoices;
}

// Register and create the server.
private static void createServer() {
  throws SMTErr, PKIErr, NetworkErr {
    SMT.AgentProxy server_samt_proxy = SMT.register(Identifiers.SERVER_ID);
    AMT.AgentProxy server_amt_proxy = AMT.register(Identifiers.SERVER_ID);
    server = new Server(server.samt_proxy, server.amt_proxy);
  }

  // Register and create the bulletin board.
  private static void createBulletinBoard() {
    throws SMTErr, PKIErr, NetworkErr {
      AMT.AgentProxy BB_proxy = AMT.register(Identifiers.BULLETIN_BOARD_ID);
      server = new Server(BB_proxy);
    }

    private static void onVote() throws SMTErr {
      int voter_id = Environment.untrustedInput();
      if (voter_id==0) // if voter casts a ballot
        voter_id = 1;
      server.onCollectBallot();
    }

    // Run the main loop of the setup.
    // First, the adversary registers his SMT and AMT
    // functionalities. Then, in a loop, the adversary decides
    // which actions are taken.
    private static void run() throws SMTErr, PKIErr, NetworkErr, AMTErr {
      Adversary adversary = new Adversary();
      while (Environment.untrustedInput() != 0 ) {
        byte[] message = Environment.untrustedInput();
        switch (decision) {
        case 0: // a voter (determined by the adversary)
          // votes according to voter's decision
          onVote();
          break;
        case 1: // server reads a message (possibly a ballot)
          // from a secure channel
          server.onCollectBallot();
          break;
        }
case 2: // the server sends the result of the election
  // (if ready) over the network
  try {
    server.onSendResult("", 1);
  } catch (NetworkError err) {}
  break;

case 3: // the server posts the result of the election
  // (if ready) on the bulletin board
  server.onPostResult();
  break;

case 4: // the bulletin board reads a message:
  BB.onPost();
  break;

case 5: // the bulletin board sends its content
  // (over the network):
  byte[] content = BB.onRequestContent();
  Environment.untrustedOutputMessage(content);
  break;

case 6: // the adversary sends a message using its
  // channel to the server
  message = Environment.untrustedInputMessage();
  adversary.channel_to_server.send(message);
  break;

case 7: // the adversary sends a message using its
  // channel to the bulletin board
  message = Environment.untrustedInputMessage();
  adversary.channel_to_BB.send(message);
  break;
}
}

public static void main(String[] args) throws SMTError, PKIError, NetworkError, AMTError {
  // the adversary determines two possible ways the voters
  // vote:
  byte[] voterChoices1 = new byte[Server.NumberOfVoters];
  byte[] voterChoices2 = new byte[Server.NumberOfVoters];
  for (int i=0; i<Server.NumberOfVoters; ++i) {
    voterChoices1[i] = (byte)Environment.untrustedInput();
    voterChoices2[i] = (byte)Environment.untrustedInput();
  }
  boolean status = select_voters_choices_and_create_voters(voterChoices1, voterChoices2);
  if (!status) return;
  create.server();
  create.bulletin.board();
  run();
}