Karlsruher Institut für Technologie

Karlsruhe Reports in Informatics 2012,8
Edited by Karlsruhe Institute of Technology,
Faculty of Informatics
ISSN 2190-4782

The KeY Approach for the Cryptographic Verification of JAVA Programs: A Case Study

Bernhard Beckert, Daniel Bruns, Ralf Küsters, Christoph Scheben, Peter H. Schmitt, and Tomasz Truderung

2012

Fakultät für Informatik

The KeY Approach for the Cryptographic Verification of JAVA Programs: A Case Study*

Bernhard Beckert[1], Daniel Bruns[1], Ralf Küsters[2], Christoph Scheben[1], Peter H. Schmitt[1], and Tomasz Truderung[2]

[1] Karlsruhe Institute of Technology
[2] Universität Trier

Abstract. In this paper, we report on an ongoing case study in which we use the KeY tool, a theorem prover for checking functional correctness and noninterference properties of JAVA programs, to establish computational indistinguishability for a simple JAVA program that involves clients sending encrypted messages over an untrusted network to a server.
The analysis uses a general framework, recently proposed by Küsters et al., which enables program analysis tools, such as KeY, that can check (standard) noninterference properties for JAVA programs to establish computational indistinguishability properties.

1 Introduction

Computational indistinguishability is a fundamental security property that can be used to express, for instance, strong secrecy of keys/messages, privacy of votes, unlinkability of communication (to prevent tracking of devices, RFID tags or contactless smartcards, and people), and as such, is relevant in many security critical applications, including secure message transmission, key exchange, anonymous communication, and e-voting. Two systems S_1 and S_2 are computationally indistinguishable if no probabilistic polynomially bounded environment (adversary) is able to distinguish, with more than negligible probability, whether it interacts with S_1 or S_2.

In [13], Küsters et al. have proposed a general framework for establishing computational indistinguishability properties for JAVA(-like) programs using program analysis tools that can check (standard) noninterference properties [7] for JAVA programs. Several such tools exist, including Joana [8], KeY [4], a tool based on Maude [1], and Jif [19,21]. However, these tools cannot deal with cryptography directly. In particular, they cannot deal with probabilities and the noninterference properties that they prove are w.r.t. unbounded adversaries, rather than probabilistic polynomially bounded adversaries. For example, if a message is encrypted and the ciphertext is given to the adversary, the tools consider this to be an illegal information flow (or a declassification), because a computationally unbounded adversary could decrypt the message. This problem has long been observed in the literature (see, e.g., [25] and references therein).

* This work was supported by the German National Science Foundation (DFG) under projects "Program-level Specification and Deductive Verification of Security Properties" (BE 2334/6-1 and SCHM 987/9-1) and "Implementation-Level Analysis of E-Voting Systems" (KU 1434/6-1) within the priority programme 1496 "Reliably Secure Software Systems – RS^3".

Now, to nevertheless enable such tools to deal with cryptography and to establish computational indistinguishability properties, the approach taken in the general framework by Küsters et al. is to use techniques from simulation-based security (see, e.g., [5,22,14]). More precisely, in order to establish computational indistinguishability properties for a JAVA program (which may describe a distributed system), the idea is to first check noninterference for the JAVA program under consideration where cryptographic operations (such as encryption) are performed within ideal functionalities. Such functionalities typically provide guarantees even in the face of unbounded adversaries and can often be formulated without probabilistic operations. These ideal functionalities can then be replaced by their realizations. Now, by theorems stated as part of the framework, from the noninterference of the JAVA program with ideal functionalities one obtains computational indistinguishability for the actual JAVA program (with the ideal components replaced by the real ones).

As further discussed in [13], language-based verification of computational indistinguishability properties is a very challenging task, especially for practical programming languages. The approach taken in [13] is new and in fact there exists almost no prior work in the literature that tackles this problem.

In [13], a first case study demonstrating the feasibility and the usefulness of the approach was carried out using the tool Joana [8], a fully automated tool for proving noninterference properties of JAVA programs.

In this paper, we report on an ongoing case study in which the KeY tool [4] is used within the framework of Küsters et al. in order to establish computational indistinguishability for a simple JAVA program that involves clients sending encrypted messages over an untrusted network to a server.

The KeY system, see [4] as the main reference, is a methodology and tool for deductive verification of annotated sequential JAVA programs. From a user's perspective it is based on the design-by-contract paradigm as advocated by Betrand Meyer in [17,18]. For the annotation of the JAVA source code an extension of the Java Modeling Language (JML) is employed, see [16]. The internal logic of the KeY system is Dynamic Logic, the basics of which will be reviewed in Subsection 3.1 below. In keeping with the design-by-contract paradigm as well as the approach taken by JML verification in the KeY system is strictly modular. That is to say, the verification of a method contract is performed without any assumptions on the situation in which a method may be called. Thus, a contract for method `m()`, once proved correct, may be applied everywhere `m()` is called. In this sence we may say that KeY adopts an *open system* perspective.

Verification of the case study with KeY is still ongoing work. The precise state of the work and the challenges to be met will be explained in Section 6. We may however already conclude that the combination of verification with KeY and the general framework is a very promision and fruitful approach.

Structure of the paper. In the next two sections, we briefly recall the general framework and the KeY approach. In Section 4, we argue that noninterference as proven in the KeY approach implies noninterference as considered in the general framework, and hence, the KeY approach delivers what is needed for the general framework. The Java program to be analyzed in our case study is presented in Section 5, with the verification process described in Section 6. We conclude in Section 7.

2 The General Framework for the Cryptographic Verification of JAVA Programs

In this section, we briefly recall the general framework for the cryptographic verification of Java programs from [13].

2.1 Jinja+: A JAVA-like language

The framework is stated for a JAVA-like language called *Jinja+*. Jinja+ is based on *Jinja* [12] and extends this language with some additional features that are useful or needed in the context of the 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`. Among the features of JAVA that are *not* covered by Jinja+ are: abstract classes, interfaces, strings, and concurrency.

Syntax of Jinja Expressions in Jinja are constructed recursively and include: (a) creation of a new object, (b) casting, (c) literal values (constants) of types boolean and int, (d) `null`, (e) binary operations, (f) variable access and variable assignment, (g) field access and field assignment, (h) method call, (i) blocks with locally declared variables, (j) sequential composition, (k) conditional expressions, (l) while loop, (m) exception throwing and catching.

A *program* or a *system* is a set of class declarations. A *class declaration* consists of the name of the class and the class itself. A *class* consists of the name of its direct superclass (optionally), a list of field declarations, and a list of method declarations, where we require that different fields and methods have different names. A *field declaration* consists of a type and a field name. A *method declaration* consists of the method name, the formal parameter names and types, the result type, and an expression (the method body). Note that there is no `return` statement, as a method body is an expression; the value of such an expression is returned by the method.

In what follows, by a *program* we will mean a complete program (one that is correct and can be executed). We assume that a program contains a unique static method `main` (declared in exactly one class); this method is the first to be called in a run. By a *system* we will mean a set of classes which is correct (can be compiled), but possibly incomplete (can use not defined classes). In particular, a system can be extended to a (complete) program.

Some construction of Jinja (and the richer language Jinja+, specified below) are illustrated by the program in Figure 1, where we use JAVA-like syntax (we will use this syntax as long as it translates in a straightforward way to a Jinja/Jinja+ syntax).

Jinja comes equipped with a type system and a notion of well-typed programs. We will consider only well-typed programs.

Semantics of Jinja Following [12], we briefly sketch the small step semantics of Jinja. The full set of rules, including those for Jinja+ (see the next subsection) can be found in [13].

A *state* is a pair of *heap* and a *store*. A *store* is a map from variable names to values. A *heap* is a map from references (addresses) to *object instances*. An *object instance* is a pair consisting of a class name and a *field table*, and a field table is a map from field names (which include the class where a field is defined) to values.

The small step semantics of Jinja is given as a set of rules of the form $P \vdash \langle e,s \rangle \rightarrow \langle e',s' \rangle$, describing a single step of the program execution (reduction of an expression). We will call $\langle e,s \rangle$ ($\langle e',s' \rangle$) a *configuration*. In this rule, P is a program in the context of which the evaluation is carried out, e and e' are expressions and s and s' are states. Such a rule says that, given a program P and a state s, an expression e can be reduced in one step to e', changing the state to s'.

Jinja+ The basis for the general framework in [13] is a language that extends Jinja with: (a) the primitive type `byte` with natural conversions from and to `int`, (b) arrays, (c) `abort` primitive, (d) static fields (with the restriction that they can be initialized by literals only), (e) static methods, (f) access modifier for classes, fields, and methods (such as `private`, `protected`, and `public`), (g) final classes (classes that cannot be extended), (h) the `throws` clause of a method declaration.

Exceptions, which are already part of Jinja, are particularly critical for the security properties we are interested in because they provide an additional way information can be transfered from one part of the program to another.

We assume that Jinja+ programs have unbounded memory. The reason for this modeling choice is that the formal foundation for the security notions used in the general framework of [13] are based on *asymptotic* security. This kind of security definitions only makes sense if the memory is not bounded, since the security parameter grows indefinitely.

Randomized programs. So far, we have considered deterministic programs. We will also need to consider randomized programs in the framework. For this purpose, Jinja+ programs may use the primitive `randomBit()` that returns a random bit each time it is used. Jinja+ programs that do no make use of `randomBit()` are (called) *deterministic*, and otherwise, *randomized*.

As already mentioned, when presenting program code, we will use JAVA-like syntax, as long as it translates in an straightforward way to the syntax of Jinja+. In this sense, the code presented in Figure 1 can be considered as a valid Jinja+ program.

Semantics and runs of Jinja+ programs. As already mentioned, the full set of rules of the small step semantics of Jinja+ can be found in [13].

Definition 1. *A* run *of a deterministic program P is a sequence of states obtained using the (small step) Jinja+ semantics from the initial configuration of the form $\langle e_0, (h_0, l_0) \rangle$, where $e_0 = C$.`main()`, for C being the (unique) class where `main` is defined, $h_0 = \emptyset$ is the empty heap, l_0 is the store mapping the static (global) variables to their initial values (if*

```
class A extends Exception {
  protected int a; // field with an access modifier
  static public int[] t = null; // static field
  static public void main() { // static method
    t = new int[10]; // array creation
    for (int i=0; i<10; i++) // loops
      t[i] = 0; // array assignment
    B b = new B(); // object creation
    b.bar(); // method invocation
  }
}
class B extends A { // inheritance
  private int b;
  public B() // constructor
    { a=1; b=2; } // field assignment
  int foo(int x) throws A { // throws clause
    if (a<x) return x+b;  // field access (a, b)
    else throw (new B()); // exception throwing
  }
  void bar() {
    try { b = foo(A.t[2]); } // static field access
    catch (A a) { b = a.a; } // exception catching @\label{example:catch}@
  }
}
```

Fig. 1. An example Jinja+ program (in JAVA-like notation).

the initial value for a static variable is not specified in the program, the default initial value for its type is used).

A randomized program induces a distribution of runs in the obvious way. Formally, such a program is a random variable from the set $\{0,1\}^{\omega}$ of infinite bit strings into the set of runs (of deterministic programs), with the usual probability space over $\{0,1\}^{\omega}$, where one infinite bit string determines the outcome of randomBit()*, and hence, induces exactly one run.*

The small step semantics of Jinja+ provides a natural measure for the *length of a run* of a program, and hence, the runtime of a program. The *length of a run of a deterministic program* is the number of steps taken using the rules of the small-step semantics. Given this definition, for a randomized program the length of a run is a random variable defined in the obvious way.

For a run r of a program P containing some subprogram S (a subset of classes of P), we define the *number of steps performed by S* or the *number of steps performed in the code of S* in the expected way. To define this notion, we keep track of the origin of (sub)expressions, i.e., the class they come from. If a rule is applied on a (sub)expression that originates from the class C, we label this step with C and count this as a step performed in C (see [13] for details).

2.2 Indistinguishability

We now recall what it means for two systems to be indistinguishable by environments interacting with those systems according to [13]. For this purpose, we first define interfaces that systems use/provide, how systems are composed, and environments. We then define the two forms of indistinguishability, namely perfect and computational indistinguishability. Since we consider asymptotic security, this involves to define programs that take a security parameter as input and that run in polynomial time in the security parameter.

Interfaces Before we define the notion of an interface, we emphasize that it should not be confused with the concept of interfaces in JAVA; we use this term with a different meaning.

An *interface I* is defined like a (Jinja+) system but where all method bodies as well as static field initializers are dropped. If I and I' are interfaces, then I' is a *subinterface* of I, written $I' \sqsubseteq I$, if I' can be obtained from I by dropping whole classes (with their method and field declarations), dropping methods and fields, dropping `extends` clauses, and/or adding the `final` modifier to class declarations. Two interfaces are called *disjoint* if the set of class names declared in these interfaces are disjoint.

If S is a system, then the *public interface of S* is obtained from S by (1) dropping all private fields and methods from S and (2) dropping all method bodies and initializers of static fields. A system S *implements* an interface I, written $S : I$, if I is a subinterface of the public interface of S. Clearly, for every system S we have that $S : \emptyset$.

We say that *a system S uses an interface I*, written $I \vdash S$, if S, besides its own classes, uses at most classes/methods/fields declared in I. We always assume that the public interface of S and I are disjoint. We note that if $I \sqsubseteq I'$ and $I \vdash S$, then $I' \vdash S$. We write $I_0 \vdash S : I_1$ for $I_0 \vdash S$ and $S : I_1$. If $I = \emptyset$, i.e., I is the empty interface, we often write $\vdash S$ instead of $\emptyset \vdash S$. Note that $\vdash S$ means that S is a complete program.

Interfaces I_1 and I_2 are *compatible* if there exists an interface I such that $I_1 \sqsubseteq I$ and $I_2 \sqsubseteq I$. Intuitively, if two compatible interfaces contain the same class, the declarations of methods and fields of this class in those interfaces must be consistent (for instance, a field with the same name, if declared in both interfaces, must have the same type). Note that if I_1 and I_2 are disjoint, then they are compatible. Systems that use compatible interfaces and implement disjoint interfaces can be composed:

Composition Let I_S, I_T, I'_S and I'_T be interfaces such that I_S and I_T are disjoint and I'_S and I'_T are compatible. Let S and T be systems such that not both S and T contain the method `main`, $I'_S \vdash S : I_S$, and $I'_T \vdash T : I_T$. Then, we say that S and T are *composable* and denote by $S \cdot T$ the *composition* of S and T which, formally, is the union of (declarations in) S and T. If the same classes are defined both in S and T (which may happen for classes not specified in I_S and I_T), then we always implicitly assume that these classes are renamed consistently in order to avoid name clashes.

Environments An environment will interact with one of two systems and it has to decide with which system it interacted (see below). Its decision is written to a distinct static boolean variable `result`. A system E is called an *environment* if it declares a distinct private static variable `result` of type `boolean` with initial value `false`. In the rest of the paper, we (often implicitly) assume that the variable `result` is unique in every JAVA program, i.e., it is declared in at most one class of a program, namely, one that belongs to the environment.

Let S be a system with $S : I$ for some interface I. Then an environment E is called an *I-environment for S* if there exists an interface I_E disjoint from I such that (i) $I_E \vdash S : I$ and $I \vdash E : I_E$ and (ii) either S or E contains `main`(). Note that E and S, as above, are composable and $E \cdot S$ is a (complete) program.

For a finite run of $E \cdot S$, i.e., a run that terminates, we call the value of `result` at the end of the run the *output of E* or the *output of the program $E \cdot S$*. For infinite runs, we define the output to be `false`. If $E \cdot S$ is a deterministic program, then we write $E \cdot S \rightsquigarrow \texttt{true}$ if the output of $E \cdot S$ is `true`. If $E \cdot S$ is a randomized program, we write $\mathsf{Prob}\{E \cdot S \rightsquigarrow \texttt{true}\}$ to denote the probability that the output of $E \cdot S$ is `true`.

The systems S_1 and S_2 *use the same interface* if (i) for every I_E, we have that $I_E \vdash S_1$ iff $I_E \vdash S_2$, and (ii) S_1 contains the method `main` iff S_2 contains `main`. Observe that if S_1 and S_2 use the same interface and we have that $S_1 : I$ and $S_2 : I$ for some interface I, then every I-environment for S_1 is also an I-environment for S_2.

Programs with security parameter As mentioned at the beginning of this section, we need to consider programs that take a security parameter as input and run in polynomial time in this security parameter. To ensure that all parts of a system have access to the security parameter, we fix a distinct interface I_{SP} consisting of (one class containing) one public static variable `securityParameter`. We assume that, in all the considered systems/programs, this variable (after being initialized) is only read but never written to. Therefore, all parts of the considered system can, at any time, access the same, initial value of this variable (see [13] for details). We denote by $P(\eta)$ a program that runs with security parameter η, i.e., a program where the variable `securityParameter` is initialized by η.

As discussed in [13], it is useful to parameterize the semantics of Jinja+ with the maximal (absolute) value integers can take. However, in this presentation we ignore this technical detail.

Perfect Indistinguishability We now recall the definition of (termination-insensitive) perfect indistinguishability from [13], which, as proved in [13], implies computational indistinguishability (see also below). We say that a deterministic program P *terminates*, if the run of P is finite.

Let S_1 and S_2 be deterministic systems such that $S_1 : I$ and $S_2 : I$ for some interface I. Then, S_1 and S_2 are *perfectly indistinguishable w.r.t. I*, written $S_1 \approx^I_{\mathsf{perf}} S_2$, if (i) S_1 and S_2 use the same interface and (ii) for every deterministic I-environment E for S_1 (and hence, S_2), for every security parameter η, it holds that if $E \cdot S_1(\eta)$ and $E \cdot S_2(\eta)$ terminate, then $E \cdot S_1(\eta) \rightsquigarrow \mathtt{true}$ iff $E \cdot S_2(\eta) \rightsquigarrow \mathtt{true}$.

Polynomially Bounded Systems As already mentioned at the beginning of this section, in order to define the notion of computational indistinguishability we need to define programs and environments whose runtime is polynomially bounded in the security parameter.

We start with the definition of almost bounded programs. These are programs that, with overwhelming probability, terminate after a polynomial number of steps. Formally, a program P with security parameter is *almost bounded* if there exists a polynomial p such that the probability that the length of a run of $P(\eta)$ exceeds $p(\eta)$ is a negligible function in η.[3]

We also need the notion of a bounded environment. The number of steps such an environment performs in a run is bounded by a fixed polynomial independently of the system the environment interacts with. Formally, an environment E is called *bounded* if there exists a polynomial p such that, for every system S such that E is an I-environment for S (for some interface I) and for every run of $E \cdot S(\eta)$, the number of steps performed in the code of E does not exceed $p(\eta)$.

If an environment E is both bounded and an I-environment for some system S, we call E a *bounded I-environment for S*.

For the cryptographic analysis of systems to be meaningful, we study systems that run in polynomial time (with overwhelming probability) with any bounded environment. Therefore, the following notion is needed: A system S is *environmentally I-bounded*, if $S : I$ and for each bounded I-environment E for S, the program $E \cdot S$ is almost bounded.

[3] As usual, a function f from the natural numbers to the real numbers is *negligible*, if for every $c > 0$ there exists η_0 such that $f(\eta) \leq \frac{1}{\eta^c}$ for all $\eta > \eta_0$. A function f is overwhelming if 1-f is negligible.

Computational Indistinguishability Having defined polynomially bounded systems and programs, we are now ready to recall the definition of computational indistinguishability of systems from [13]. We begin with the notion of computationally equivalent programs.

Let P_1 and P_2 be (complete, possibly probabilistic) programs with security parameter. Then P_1 and P_2 are *computationally equivalent*, written $P_1 \equiv_{\mathsf{comp}} P_2$, if

$$|\mathsf{Prob}\{P_1(\eta) \rightsquigarrow \mathtt{true}\} - \mathsf{Prob}\{P_2(\eta) \rightsquigarrow \mathtt{true}\}|$$

is a negligible function in the security parameter η.

Let S_1 and S_2 be environmentally I-bounded systems. Then S_1 and S_2 are *computationally indistinguishable w.r.t. I*, written $S_1 \approx^I_{\mathsf{comp}} S_2$, if S_1 and S_2 use the same interface and for every bounded I-environment E for S_1 (and hence, S_2) we have that $E \cdot S_1 \equiv_{\mathsf{comp}} E \cdot S_2$.

We point out that in the above definition two cases can occur: (1) `main()` is defined in E or (2) `main()` is defined in both S_1 and S_2. In the first case, E can freely create objects of classes in the interface I (which is a subset of classes of S_1/S_2) and initiate calls. Eventually, even in case of exceptions, E can get back control (method calls return a value to E and E can catch exceptions if necessary), unless S_1/S_2 uses `abort`. The kind of control E has in the case (2), heavily depends on the specification of S_1/S_2. This can go from having as much of control as in case (1) to being basically a passive observer. For example, `main()` (as specified in S_1/S_2) could call a method of E and from then on E can use the possibly very rich interface I as in case (1). The other extreme is that I is empty, say, so E cannot create objects of (classes of) S_1/S_2 by itself, only S_1/S_2 can create objects of (classes of) E and of S_1/S_2. Hence, S_1/S_2 has more control and can decide, for instance, how many and which objects are created and when E is contacted. Still even in this case, if so specified, S_1/S_2 could give E basically full control by callback objects. (As a side note, illustrating the richness of the interfaces, compared to Turing machine models, E could also extend classes of S_1/S_2 and by this, if not properly protected, might get access to information kept in these classes.)

2.3 Simulatability

We now recall what it means for a system to realize another system, in the spirit of the simulation-based approach. In a nutshell, the definition says that the (real) system R realizes an (ideal) system F if there exists a simulator S such that R and $S \cdot F$ behave almost the same in every bounded environment.

Definition 2 (Strong Simulatability [13]). *Let $I_{in}, I_{out}, I_E, I_S$ be disjoint interfaces. Let F and R be systems. Then R* realizes *F* w.r.t. the interfaces *I_{out}, I_{in}, I_E,* and *I_S, written $R \leq^{(I_{out}, I_{in}, I_E, I_S)} F$ or simply $R \leq F$, if i) $I_E \cup I_{in} \vdash R : I_{out}$ and $I_E \cup I_{in} \cup I_S \vdash F : I_{out}$, ii) either both F and R or neither of these systems contain the method `main()`, iii) R is an environmentally I_{out}-bounded system (F does not need to be), and iv) there exists a system S (the simulator) such that S does not contain `main()`, $I_E \vdash S : I_S$, $S \cdot F$ is environmentally I_{out}-bounded, and $R \approx^{I_{out}}_{\mathsf{comp}} S \cdot F$.*

The intuition behind the way the interfaces between the different components (environment, ideal and real functionalities, simulator) are defined is as follows: Both R and F provide the same kind of functionality/service, specified by the interface I_{out}. They may require some (trusted) services I_{in} from another system component and some services from an (untrusted) environment, for example, networking and certain other libraries. In addition, the ideal functionality F may require services from the simulator S, which in turn may require services from the environment. As discussed in [13], we note that the interfaces can be very rich—they allow for communication and method calls in both directions.

The notion of strong simulatability, as introduced above, enjoys important basic properties, namely, reflexivity and transitivity, and allows to prove a fundamental composition theorem (see [13] for more details).

In the case study presented in Section 5, we will analyze a system that uses public-key encryption. In this analysis, we will use an ideal functionality for public-key encryption, denoted by IdealPKE, which is provided in [13]. In [13], it was also proved that this functionality can be realized by a system called RealPKE that implements, in the obvious way, an IND-CCA2-secure public-key encryption scheme.

2.4 From Perfect to Computational Indistinguishability

Following [13], we now present results stating that if two systems that use an ideal functionality are perfectly indistinguishable, then these systems are computationally indistinguishable if the ideal functionality is replaced by its realization. This is a central step in enabling program analysis tools that cannot deal with cryptography and probabilistic polynomially bounded adversaries to establish computational indistinguishability properties.

The proof of the statement sketched above is done via two theorems in [13]. The first says that if two systems that use an ideal functionality are computationally indistinguishable, then they are also computationally indistinguishable if the ideal functionality is replaced by its realization.

Theorem 1 ([13]). *Let I, J, I_E, I_S, and I_P be disjoint interfaces with $J \sqsubseteq I_P \cup I$. Let F, R, P_1, and P_2 be systems such that (i) $I_E \cup I \vdash P_1 : I_P$ and $I_E \cup I \vdash P_2 : I_P$, (ii) $R \leq^{(I, I_P, I_E, I_S)} F$, in particular, $I_E \cup I_P \vdash R : I$ and $I_E \cup I_P \cup I_S \vdash F : I$, (iii) P_1 contains $\mathtt{main}()$ iff P_2 contains $\mathtt{main}()$, (iv) not both P_1 and F (and hence, R) contain $\mathtt{main}()$, (v) $F \cdot P_i$ and $R \cdot P_i$, for $i \in \{1,2\}$, are environmentally J-bounded. Then, $F \cdot P_1 \approx^J_{\mathsf{comp}} F \cdot P_2$ implies $R \cdot P_1 \approx^J_{\mathsf{comp}} R \cdot P_2$.*

The next theorem links perfect indistinguishability with computational indistinguishability.

Theorem 2 ([13]). *Let I be an interface and let S_1 and S_2 be deterministic, environmentally I-bounded programs such that $S_i : I$, for $i \in \{1,2\}$, and S_1 and S_2 use the same interface. Then, $S_1 \approx^I_{\mathsf{perf}} S_2$ implies $S_1 \approx^I_{\mathsf{comp}} S_2$.*

By combining Theorem 1 and Theorem 2, one immediately obtains the desired result explained above.

Corollary 1 ([13]). *Under the assumption of Theorem 1 and moreover assuming that $P_1 \cdot F$ and $P_2 \cdot F$ are deterministic systems, it follows that $P_1 \cdot F \approx^J_{\mathsf{perf}} P_2 \cdot F$ implies $P_1 \cdot R \approx^J_{\mathsf{comp}} P_2 \cdot R$.*

2.5 Perfect Indistinguishability and Noninterference

We now recall the statement from [13] that perfect indistinguishability and noninterference are equivalent for an appropriate class of systems. In combination with Corollary 1 this means that it suffices for tools to analyze systems that use an ideal functionality w.r.t. noninterference in order to get computational indistinguishability for systems when the ideal functionality is replaced by its realization.

The (standard) noninterference notion for confidentiality [7] requires the absence of information flow from high to low variables within a program. In [13], noninterference is defined for a (Jinja+) program P with some static variables x of primitive types that are labeled as high. Also, some other static variables of primitive types are labeled as low. We say that $P[x]$ is a *program with high and low variables*. By $P[a]$ we denote the program P where the high variables x are initiated with values a and the low variables are initiated as specified in P. We assume that the lengths of x and a are the same and a contains values of appropriate types; in such a case we say that a is valid. Now, noninterference for a (deterministic) program is defined as follows, where, for ease of presentation, the definition is slightly simplified compared to [13].

Definition 3 (Noninterference for Jinja+ programs [13]). *Let $P[x]$ be a program with high and low variables. Then, $P[x]$* has the noninterference property *if the following holds: for all valid a_1 and a_2, if $P[a_1]$ and $P[a_2]$ terminate, then at the end of these runs, the values of the low variables are the same.*

The above notion of noninterference deals with complete programs (closed systems). The definition can be lifted to open systems as follows

Definition 4 (Noninterference in an open system [13]). *Let I be an interface and let $S[x]$ be a (not necessarily closed) deterministic system with a security parameter, high and low variables, and such that $S : I$. Then, $S[x]$ is I-*noninterferent *if for every deterministic I-environment E for $S[x]$ and every security parameter η noninterference holds for the system $E \cdot S[x](\eta)$, where the variable `result` declared in E is considered to be a low variable.*

Now, equivalence of this notion and perfect indistinguishability follows easily by the definitions of I-noninterference and perfect indistinguishability:

Theorem 3. *[13] Let I and $S[x]$ be given as in Definition 4 with no variable of S labeled as low (only the variable `result` declared in the environment is labeled as low). Then the following statements are equivalent:*
(a) For all valid a_1 and a_2, we have that $S[a_1] \approx^I_{\mathsf{perf}} S[a_2]$.
(b) I-noninterference holds for $S[x]$.

As already mentioned, in combination with Corollary 1 this theorem reduces the problem of checking computational indistinguishability for systems that use real cryptographic schemes to checking noninterference for systems that only use ideal functionalities.

2.6 A Proof Technique for Noninterference in Open Systems

There are many tools that can deal with classical noninterference (noninterference of a complete program), including the KeY tool. In our case study, we want to apply the KeY tool to prove noninterference in certain open systems. In [13], we have developed techniques that help with this process. We shortly recall these techniques; for more details, see [13].

Our technique works for the following cases. We assume that a system S communicates with its environment E through an interface I_E provided by E (where E does not use the public interface of S) such that S and E exchange information through values of primitive types, arrays of primitive types, simple objects, and throwing exceptions. Some additional restrictions are imposed on the system S. These restrictions (formally specified in [13]) guarantee that, although references are exchanged between E and S, the communication resembles exchange of pure data.

Now, in order to show I-noninterference for S, with $I = \emptyset$ (the environment E does not use the public interface of S), we use the following technique: Given S and I_E as above, we consider a specific system $E^*_{I_E}$ that implements the interface I_E (see the definition of this system in [13]). This system is not closed: it uses the interface I_{IO} consisting of (a class with) two static methods:

```
public static void unstrustedOutput(int x);
public static int untrustedInput();
```

We obtain the following fact, which is a corollary of Theorem 6 in [13]:[4]

Corollary 2. *Let S be as above and $I = \emptyset$. Then, S is I-noninterferent if the system $E^*_{I_E} \cdot S$ is I-noninterferent.*

Note that the interface I_{IO} used by the system $E^*_{I_E} \cdot S$ is very restricted which facilitates the verification process described in the following sections. In short, the formulation of contracts for unspecified methods is greatly simplified.

We note that the program P considered in our case study (Section 5) falls into the family of programs that can be handled by the technique sketched above.

3 The KeY Approach

In the following we present the KeY approach to verify information flow properties. The presentation follows [27].

As a starting point, some basics on Java Dynamic Logic (JAVADL) will be summarised in the next section. Afterwards, a simple definition of non-interference and its formalisation in JAVADL will be considered in Section 3.2. The formalisation will be illustrated on a password checker example which will be used and extended throughout the presentation of the KeY approach. Section 3.3 gives a short introduction to JML and JML* and continues with the definition of an extension of JML* suitable for the specification of information flow properties. Finally Section 3.4 describes the translation of the JML* extensions into JAVADL.

[4] Theorem 6 in [13] makes a stronger statement in that only specific implementations of the interface I_{IO} are considered.

3.1 Basics on JAVA Dynamic Logic

Dynamic Logic is an extension of typed first-order logic tailored towards reasoning about computer programs, see [9] for an early publication and [10] for a modern account. Typical formulas, that go beyond first-order logic, are of the form $\langle\pi\rangle F$ or $[\pi]F$ where F is again a Dynamic Logic formula and π is a program. In theoretical investigations the programs π are taken from some abstract programming language. In the instantiation of Dynamic Logic that we are concerned with, JAVADL, π can be an arbitrary, executable, sequential JAVA program. More formally, dynamic logic is a multimodal logic in which there are modalities $\langle\pi\rangle$ and $[\pi]$ for every program π. Dynamic logic extends Hoare logic: The Hoare triple $\{\phi\}\pi\{\psi\}$ can be expressed in dynamic logic as $\phi \rightarrow [\pi]\psi$.

For the definition of terms we implicitly assume that a vocabulary $\Sigma = Rel \uplus Fun \uplus LV$ of relation and functions symbols and a set of logical variables has been fixed. Among the function symbols we distinguish the subset $\mathscr{R} \subseteq Fun$ of *reference symbols*. Reference symbols are

1. all instance and static fields as well as
2. `this`, `result` and method parameters.

Expressions built up exclusively from symbols in $\mathscr{R}$ are called *reference expressions*. We use $Exp_{\mathscr{R}}$ to stand for the set of all reference expressions. The semantics of Dynamic Logic is based on the notion of a program state, i.e., an assignment of values to all program variables, global and local. The formula $\langle\pi\rangle F$ is true in state s, in symbols $val_s([\pi]F) = tt$, if the program π started in s terminates and formula F is true in the terminating state. This corresponds to total correctness assertions in Hoare logic that the reader might be more familiar with. Dually $[\pi]F$ is true in state s, if either π does not terminate when started in s or $\langle\pi\rangle F$ is true in s. This corresponds to partial correctness assertions in Hoare logic. The first-order logic part of JAVADL contains types $Heap$ and $Field$ and thus allows quantifications $\forall Heap\, h$ and $\forall Field\, f$. Furthermore there is an implicit program variable *heap* of type $Heap$ that evaluates in any state to the current heap of the JAVA program. The values of fields, arrays and information on created objects are stored and accessed by suitable functions as formalized in the theory of abstract arrays, see [23, pages 69 – 70] and [28]. The details of this model play no role in the current report. Thus, the current value of the implicit program variable *heap* determines completely the current state of the JAVA heap.

JAVADL uses an additional modal operator $\{v := t\}$ called an *update*, where v is a program variable and t a JAVADL expression. A formula $\{v := t\}F$ is true in state s if F is true in the state s' with $s'(w) = s(w)$ for variables $w \neq v$ and $s'(v)$ equals the value of expression t in state s. Updates serve more than one purpose in JAVADL. They are ultimately necessary for an axiomatization of forward symbolic execution. For the reader of this report it suffices to think of updates as an interface between logical and program variables. While program variables may occur in formulas, logical variables are not allowed in programs. But logical variables may occur in the expression t in an update $\{v := t\}$. By $\{v_1 := t_1 \,||\, v_2 := t_2\}$ we denote parallel composition of the updates $\{v_1 := t_1\}$ and $\{v_2 := t_2\}$.

3.2 Information Flow and JAVA Dynamic Logic

The most prominent information flow property is *non-interference*. In the simple case of theoretical programming languages non-interference is defined for a program P and a partition of the program variables of P in low security variables low and high security variables $high$. The low variables are publicly readable variables whereas the $high$ variables contain secret data which should be protected. Intuitively non-interference expresses that there should be no data-flow from $high$ variables into low ones. In this report we classify reference symbols as *high* or *low*. A reference expression is called *high* (*low*) if its leading reference symbol is *high* (*low*). By $\mathscr{L}$ we denote the set of all *low* reference expressions.

Definition 5 (Low equivalence). *We call two states s_1, s_2* low equivalent*, with respect to $\mathscr{L}$, in symbols $s_1 \sim_{\mathscr{L}} s_2$, iff $val_{s_1}(e) = val_{s_2}(e)$ for all $e \in \mathscr{L}$.*

Definition 6 (Simple Non-interference). *We say that there is no interference of high references with low references in P iff*
for all states s_1, s_2, s'_1, s'_2 such that program P started in s_i terminates in s'_i
the condition $s_1 \sim_{\mathscr{L}} s_2 \Rightarrow s'_1 \sim_{\mathscr{L}} s'_2$ holds.

Simple non-interference can be formulated naturally in JAVA Dynamic Logic with the help of self-composition as it will be shown in the next section.

Formalising Simple Non-Interference in JAVADL Simple non-interference will be expressed in JAVADL with the help of self-composition [2,6]. In short, self-composition executes a program twice on any two low equivalent states and compares whether the results are also low equivalent. In our formulation we replace the program P from Definition 6 by a call to a method m of class C with parameters $p_1, \dots, p_n$ of type $\mathscr{T}_1, \dots, \mathscr{T}_n$ and a return value of type $\mathscr{T}_r$.

Lemma 1. *Let $\mathscr{L}$ be all reference expressions occurring in method m with leading symbol classified as low. If the formula*

$$\begin{aligned}
&\forall Heap\, s^1_{in}, s^2_{in}, s^1_{out}, s^2_{out} \\
&\forall C\, this\, \forall \mathscr{T}_1\, p^1_1\ \dots\ \forall \mathscr{T}_n\, p^1_n\, \forall \mathscr{T}_r\, r^1\, \forall \mathscr{T}_1\, p^2_1\ \dots\ \forall \mathscr{T}_n\, p^2_n\, \forall \mathscr{T}_r\, r^2\ \Big(\\
&\qquad\qquad this \neq null \\
&\qquad\qquad \wedge\ \mathscr{U}^1_{in} \langle \mathscr{T}_r\ result = this.m(p_1, \dots, p_n) \rangle (h^1_{out} = heap \wedge r^1 = result) \\
&\qquad\qquad \wedge\ \mathscr{U}^2_{in} \langle \mathscr{T}_r\ result = this.m(p_1, \dots, p_n) \rangle (h^2_{out} = heap \wedge r^2 = result) \\
&\qquad\qquad \wedge\ \bigwedge_{e \in \mathscr{L}} \mathscr{U}^1_{in}\, e = \mathscr{U}^2_{in}\, e \\
&\qquad\qquad \rightarrow\ \bigwedge_{e \in \mathscr{L}} \mathscr{U}^1_{out}\, e = \mathscr{U}^2_{out}\, e \Big)
\end{aligned} \tag{1}$$

with Updates $\mathscr{U}^i_x$ defined as $\mathscr{U}^i_x = \{heap := s^i_x \,||\, p_1 := p^i_1 \,||\, \dots \,||\, p_n := p^i_n \,||\, r := r^i\}$ is valid then there is no information flow from the high locations to the low ones in m.

Proof. See [27].

```
1  class PasswordFile {                                              high symbols
2    private int[] names, passwords;
3    //@ invariant names.length == passwords.length;
4    /*@ normal_behavior
5      @   ensures     \result ==
6      @               (\exists int i; 0<=i && i<names.length;
7      @               names[i]==user && passwords[i]==password);
8      @   accessible  names, names[*], passwords, passwords[*];
9      @   modifies    \nothing;                                           @*/
10   public boolean check(int user, int password) {
11     /*@ loop_invariant  0 <= i && i <= names.length &&
12       @                ( \forall int j; 0 <= j && j < i;
13       @                  !(   names[j]==user
14       @                    && passwords[j]==password) );
15       @ assignable  \nothing;
16       @ decreases   names.length - i;                                  @*/
17     for (int i = 0; i < names.length; i++) {
18       if (names[i] == user && passwords[i] == password) {
19         return true;
20       }
21     }
22     return false;
23   }
24 }                                                                  low symbols
```

Fig. 2. Example of a password checker in JAVA with a full functional JML-specification and an informal annotation for *low*- and *high*-reference symbols.

Formula (1) has been inspired by [6, Formula (7)]. However, the formalisation is still quite abstract and can't be used directly as proof obligation for the KeY System. Some details on general assumptions like invariants and the wellformedness of the heaps etc. are abstracted away. Furthermore, it does not cover important features of a practicable non-interference specification language for JAVA, as discussed in Section 3.4. Before we introduce the new program-level specification language for non-interference in Section 3.3, we want to illustrate the formalisation of simple non-interference with an example.

Example The frequently used password checker example will be used to illustrate the formalisation. The example will be extended throughout the presentation of the KeY approach. The considered implementation (Figure 2) consists of a class `PasswordFile` with two private arrays, `names` and `passwords`, which store the user-names and their corresponding passwords at the same index. Obviously, the length of those two arrays has to coincide. This is formulated with the help of an JML-invariant in line 3. For the moment the reader may assume that such an invariant holds in any state of the program. More details on JML will be given in Section 3.3. Furthermore, the class contains a

```
∀ Heap h_in_1, h_in_2, h_out_1, h_out_2  // independend heaps
∀ PasswordFile this                      // considered class
∀ int user1, password1, user2, password2 // method arguments
∀ boolean result1, result2               // return values
// General Assumtions + Class Invariants
    wellFormed(h_in_1) ∧ wellFormed(h_in_2) ∧ ...
// Independent Symbolic Executions
  ∧ {heap := h_in_1}\[{ ...
      boolean r = this.check(user1 ,password1)@PasswordFile;
    ...}\]( h_out_1 = heap ∧ result1 = r )
  ∧ {heap := h_in_2}\[{ ...
      boolean r = this.check(user2 ,password2)@PasswordFile;
    ...}\]( h_out_2 = heap ∧ result2 = r )
// Comparision of the low variables
  ∧ user1 = user2 ∧ password1 = password2
  → result1 = result2
```

Fig. 3. Formalisation of non-interference in JAVADL for the example of Figure 2. The three dots "..." mark passages where some less important JAVADL details have been abstracted away.

method `check` which takes a user-name and a password. It checks whether there exists an index *i* at which the array `names` contains the user-name and at which the array `passwords` contains the password. If such an index exists, the method returns true, otherwise false. The implementation covers a full functional JML-specification consisting of a method contract and a loop-invariant. Those specifications are not relevant at the moment, but will be discussed in Section 3.3. Still, a formal specification of *low* and *high* reference symbols is missing, since the current version of JML does not allow for such specifications. Let's assume the arrays `names` and `passwords` and their contents are considered as *high* whereas the parameters `user` and `password` as well as the not explicitly named `return`-variable are considered as *low*. This is reasonable since user-names and passwords normally should be kept secret whereas the caller knows the user-name and password he entered as well as the returned value. The method *check* translates in connection with these informal *low* and *high* specifications to the JAVADL formula of Figure 3. The formula is assembled as follows: the first part contains some general assumptions, which have been abstracted away in Formula (1) and are not central here either. The following two parts contain the symbolic execution and comparison as introduced in Section 3.2. Still, there is a slight difference: in this particular example only the return values have to be compared, because the heap does not contain low reference symbols and the values of the parameters are not observable after the return of the method.

In the next section, it will be shown how JAVA programs can be annotated systematically with non-interference specifications.

3.3 Program-Level Specifications

The last section showed how, in principle, non-interference can be formalised in JAVADL and how proof obligations can be generated manually out of JAVA programs with an informal annotation of *low* and *high* reference symbols. This section will discuss how JAVA programs can be annotated with non-interference specifications which seamlessly integrate with functional specifications in JML* and which are suitable for automatic translation into JAVADL. The specification entities are in particular suitable for the (implicit) specification of security lattices and intentional information leakage. Before the new specification entities are introduced in Section 3.3, the next section will present some basics on JML and its dialect JML*. Section 3.3 illustrates the entities on the example of Figure 2 thereafter.

JML and JML* The Java Modeling Language (JML) is a popular language for the behavioral specification of JAVA code [15]. It adopts the design by contract (DBC) methodology. JAVA expressions enriched with other specification constructs such as quantifiers are used to write assertions, such as pre- and postconditions and invariants. [28] introduced a dialect of JML, called JML*, which is suitable for modular specifications. Our approach will be based on this dialect. In the following, the specification entities which are most important in the context of this work will be explained shortly by the example of Figure 2. These entities are method contracts, invariants and model fields.

The method contract of Figure 2 starts with the keyword `normal_behavior`. This means that the method won't throw an exception if the precondition of the method is fulfilled. Preconditions are specified via the keyword `requires`. If the keyword is missing—as it is the case in our example—the precondition is implicitly defined as `true`. Thus, the specification guarantees that no exception will be thrown. Postconditions are specified via the keyword `ensures`. In Figure 2, lines 5 to 7 specify the postcondition of the method `check`, which says that the result of the method is `true` iff there exists an index *i* at which the array `names` equals the value passed by the `user` parameter and at which the array `passwords` equals the value passed by the parameter `password`. Furthermore, the keyword `modifies` defines a set of heap locations whose values may be changed at most by the execution of the method. In our case, no locations may be changed. Similarly, the keyword `accessible` defines a set of heap locations whose values may be read at most by the execution of the method. Here, the specification of `check` expresses that at most the heap locations of the fields `names` and `passwords` as well as the entries of the two arrays are read. Line 3 shows an invariant specification. It says that the lengths of the arrays `names` and `passwords` coincide. In this work it can be assumed that invariants hold in every state of the program execution. The real semantics is more complicated, but it is not essential to understand the semantics in detail in order to follow the presentation of this work. Finally, model fields have to be considered. The following example shows a model field definition:

```
/*@ model \refset pwdFileManager;
  @ represents pwdFileManager =
  @     names, names[*], passwords, passwords[*];
```

```
@*/
```

The first line declares the model field `pwdFileManager` of type `\refset`. Lines two and three define the set of locations, to which the model field evaluates. In the context of this work we consider only model fields of type `\refset`.

Next, the concepts behind the new JML* specification entities are introduced.

Concepts for the JML* Extensions and their Motivation Given the definition of non-interference from Section 3.2, one of the first questions to ask is: how can *low* and *high* reference symbols be defined in JML*? We will provide in this section an answer to a more general question: How can on the level of JML* reference expressions be classified with respect to security levels? For practicable information flow analysis a classification into *low* and *high* symbols is in many cases found to be too coarse. Most existing analysis tools use lattices of security levels instead (see for instance [20]). In a security lattice information may flow only from lower levels to higher ones. The power set of all reference expressions $\mathscr{P}(Exp_{\mathscr{R}})$ with set union and set intersection as operations forms the most general lattice of security types [11,6]: any other type lattice is subsumed by it. Therefore it is reasonable and quite common to restrict oneself to (sublattices of) $\mathscr{P}(Exp_{\mathscr{R}})$. Given a security lattice, a security policy is defined as a mapping of the set of reference expressions to the set of security levels of the lattice.

The probably most straight forward approach to define a security policy in JML would be to require the specifier (1) to provide a security lattice and (2) to annotate the declaration of each field, parameter, return value etc. with a security level from this lattice. However, we favour another concept which concentrates on the specification of the knowledge which the actors of a system may have about a system. Every specification in our approach implies a security policy and vice versa, as shown in [27]. We think that specifications on the knowledge of actors can be deduced easier from high-level security requirements than a suitable security policy. Furthermore, our approach defines the implied security policies in a decentralised way which is helpful for modular specifications. We will illustrate our approach with the help of the following banking example. In this subsection we will use UML diagrams to motivate our approach. This will help us to abstract away from program level details that would only blur the big picture. The technical definitions in Subsection 3.3 will of course be again on the JAVA level.

Figure 4 shows a use-case diagram and a class diagram for a banking example. The actors in this example are bank-customers and bank-employees. Customers can view the balance of (their) accounts and draw money while employees may see the balance of (all) accounts and create new accounts. A reasonable security requirement is that a customer may know at most the data belonging to his accounts while an employee may know everything except the passwords of the accounts. This requirement is illustrated in the object diagram of Figure 5 for three customers and one employee. Each kind of smiley represents an actor and marks the fields which are allowed to be observed / known by this actor. In the following these sets of reference expressions are called *views*. The lower part of Figure 5 summarizes these requirements at the level of a class diagram.

Definition 7 (View). *A view V is an expression which evaluates to a set of reference expressions.*

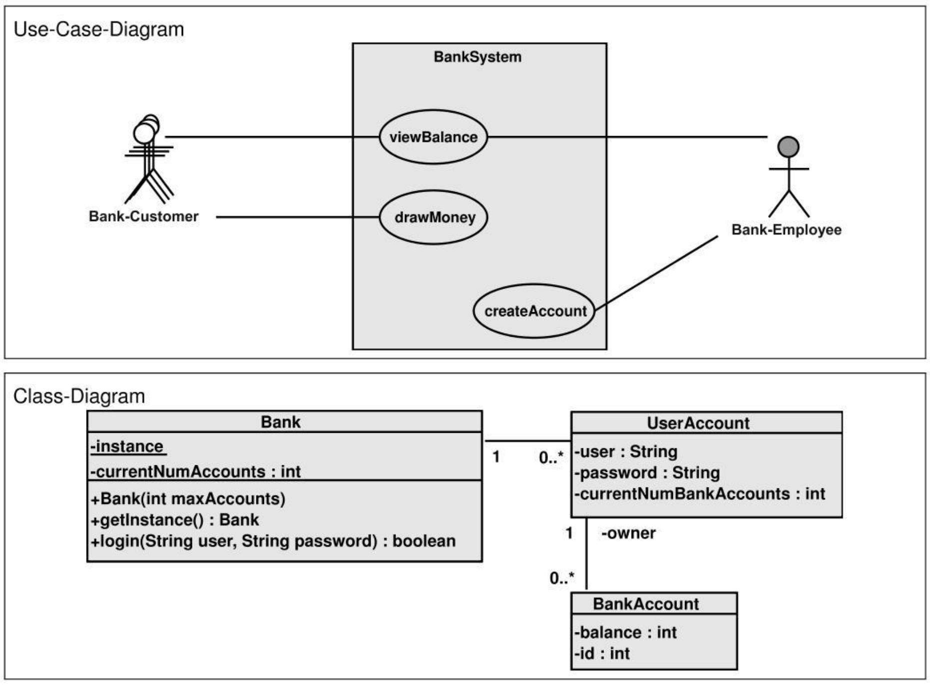


Fig. 4. A use-case diagram and a class diagram for a banking scenario.

These expressions may contain, as in the example in Figure 5, set union and the iterator symbol *. The symbols `userAccounts` and `bankAccounts` denote the shown associations between the classes `Bank` and `UserAccount` in the first case and between `UserAccount` and `BankAccount` in the second. On the programming language level these could be implemented as fields of type `UserAccount[]` and `BankAccount[]`.

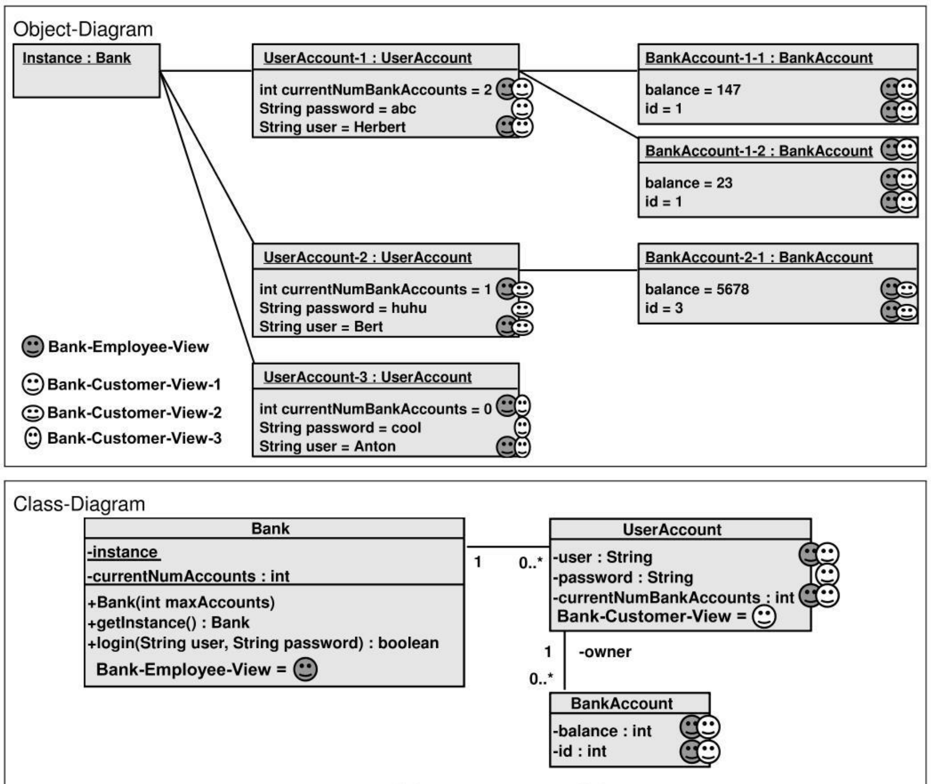
Non-interference can be defined for a set of views in the obvious way:

Definition 8 (Low equivalence for views). *We call two states s_1, s_2* low equivalent *with respect to a view V, in symbols $s_1 \sim_V s_2$, iff $val_{s_1}(V) = val_{s_2}(V)$ and $val_{s_1}(e) = val_{s_2}(e)$ for all $e \in val_{s_1}(V)$.*

Allowing model fields in expression for views one may define a view that e.g., evaluates to a set of expressions for all elements in a linked list. As a consequence $val_s(V)$ depends on s. This motivates the first equation on the left hand side of the previous definition.

Definition 9 (Non-interference for a set of views). *We say that there is no interference of higher references with lower references with respect to the security policy defined by a set of views $\mathcal{V}$ in P iff*
for all views $V \in \mathcal{V}$ and all states s_1, s_2, s_1', s_2' such that program P started in s_i terminates in s_i' the condition $s_1 \sim_V s_2 \Rightarrow s_1' \sim_V s_2'$ holds.

In the next section the JML* extensions for non-interference specifications are introduced.



$$
\begin{aligned}
\text{Bank-Employee-View} = {} & userAccounts[*] \cup userAccounts[*].user \\
& \cup userAccounts[*].currentNumBankAccounts \\
& \cup userAccounts[*].bankAccounts[*] \\
& \cup userAccounts[*].bankAccounts[*].balance \\
& \cup userAccounts[*].bankAccounts[*].id \\
\text{Bank-Customer-View} = {} & user \cup password \cup currentNumAccounts \\
& \cup bankAccounts[*] \cup bankAccounts[*].balance \cup bankAccounts[*].id
\end{aligned}
$$

Fig. 5. An object diagram to the class diagram of Figure 4 with annotated views and the class diagram with the same set of views.

Extending JML* for Non-Interference Specifications Our JML* extension provides the possibility to specify for every method a set of views (which defines the security policy) which is respected by that method. More precisely, every method contract can be extended by a `respects`-clause which defines the set of views:

```
/*@ respects    \set_union(names, names[*]),
  @             \set_union(passwords, passwords[*]);
  @*/
```

```
boolean check(int user, int password) { ...
```

The respects-clause is a usual clause in a method contract and takes a list of views as an argument. Views evaluate to sets of reference expressions and list concatenation is interpreted as set union. A contract may contain multiple respects-clauses. In this case the clauses are treated as one big respects-clause consisting of the concatenation of all listed location sets. The semantics of the clause is that in case the precondition of the contract evaluates to true, the method fulfills non-interference for the defined set of views as formalised in Definition 9.

Sets of views can, but don't have to be specified explicitly. An alternative is to use a model field that can be reused in different contexts. The model field may remain underspecified or its value may be given by a represents-clause.

```
//@ model \refset anyUser;
//@ respects anyUser;
boolean check( ... ) { ...
```

The above example illustrates an underspecification. The model field anyUser can represent any reference set. This means in particular that every singleton set is respected by the method which in turn means that no information flows at all.

Because we aim at information flow specifications which are modular at the method level, a security level has also to be specified for every parameter and the return value of a method. This is illustrated in the following example:

```
public int low;
private int high;
//@ respects low;
int m(int param) {
  low = param; return param;
}
```

Imagine low and high are *low* and *high* fields, respectively, and the method m is called with high as parameter. Then the *high* value will be assigned to low and therefore the call would be unsafe. On the other hand, if m is called with low as parameter everything is fine. In order to assign parameters and the return-value a security level, the parameter-names and the \return-statement are \refset-expressions and can be used in the respects-clause.

The respects-clause is sufficient to specify non-interference for methods in JML*. However, as it is well known (see for instance [20]), non-interference on its own is too restrictive; many useful JAVA programs contain intentional information leaks. This includes the program of Example 2. KeY won't be able to show the universal validity of the formula of Figure 3 because the method check indeed leaks some information about the secret arrays names and passwords: the information whether the passed username and password are contained in names and passwords or not. In order to distinguish intentional information leaks from unintentional ones, intentional leaks have to be specified clearly. Those specifications are called declassifications [26]. In our case, declassifications specify the information which is allowed to leak in form of terms. The value of the terms in the pre-state of the execution of a method is the information which is allowed to leak. In the case of Example 2 this is the Boolean term

```
\exists int i; 0 <= i && i < names.length;
```

```
names[i] == user && passwords[i] == password
```

Following the discussion e.g., in [2, end of Section 6] and [26, Section 2.1] the meaning of declassification is formalized by a further restriction on the input equivalence in Definition 9, leading to:

Definition 10 (Non-interference for a set of views with declassification). *We say that there is no interference of higher references with lower references with respect to the security policy defined by a set of views $\mathcal{V}$ except for declassification t in P iff for all views $V \in \mathcal{V}$ and all states s_1, s_2, s_1', s_2' such that program P started in s_i terminates in s_i' the condition $s_1 \sim_V^t s_2 \Rightarrow s_1' \sim_V s_2'$ holds, where $s_1 \sim_V^t s_2 \Leftrightarrow s_1 \sim_V s_2$ and $val_{s_1}(t) = val_{s_2}(t)$.*

On the JML* level declassification is specified by the `declassify`-clause. We present here a generalisation of declassification suggested by [26] that goes beyond Definition 10. The following example illustrates the usage of this general `declassify`-clause.

```
/*@ declassify
  @   ( \exists int i; 0 <= i && i < names.length;
  @     names[i] == user && passwords[i] == password )
  @   \from pwdFileManager  \to \result  \if true;
  @*/
boolean check(int user, int password) { ...
```

A declassification in its full generality is a tuple $(D_{term}, D_{from}, D_{to}, D_{if})$, where D_{if} is a Boolean expression, D_{to} and D_{from} are views and D_{term} is an expression. The semantics is the following: for every view V and every pair s_1, s_2 of pre-states of an method invocation, if the preconditions (1) $val_{s_1}(D_{if}) = val_{s_2}(D_{if}) = true$ (the condition of the `\if`-part D_{if} is fulfilled in the pre-states of the execution), (2) $val_{s_1}(V) \subseteq val_{s_1}(D_{to}) \wedge val_{s_2}(V) \subseteq val_{s_2}(D_{to})$ (the view V is a subset of the `\to`-part D_{to}) and (3) for all states s, s' with $val_s(D_{from}) = val_{s'}(D_{from}) \wedge \forall e \in val_s(D_{from})\ (val_s(e) = val_{s'}(e))$ the condition $val_s(D_{term}) = val_{s'}(D_{term})$ holds (the declassified term D_{term} depends only on the references in the `\from`-part D_{from}) hold, then $val_{s_1}(D_{term}) = val_{s_2}(D_{term})$ will be true.

The defaults for the `\from`-, `\to`- and `\if`-parts are `\everything`, `\everything` and `true`, respectively.

Example Figure 6 gives a complete example of the specification of the `check` method from Figure 2. The specification of Figure 2 is extended in two parts. The first part, line 1, declares an underspecified model field `anyUser`, as discussed in Section 3.3. The KeY system contains a built-in type `\locset` for *location sets*. Locations are those references that are stored on the heap. The model field `anyUser` can thus stand for any location set. The second part, lines 9 to 12, extends the method contract of the method `check` by a non-interference specification. The `respects`-clause states that the views `anyUser` and $\{\backslash result\}$ can't learn anything through the execution of `check`. Since `anyUser` can stand for any set of references on the heap, the first part of the specification states that no information may flow on the heap. The second part, the view $\{\backslash result\}$, states that the result may depend at most on itself. Finally, the `declassify`-clause states that the information whether the passed user-name and password are contained in `names` and `passwords` or not may be leaked to the result of the method.

```
/*@ model \locset anyUser;
  @
  @ normal_behavior
  @   ensures     \result ==
  @               ( \exists int i; 0<=i && i<names.length;
  @                names[i]==user && passwords[i]==password);
  @   accessible  names, names[*], passwords, passwords[*];
  @   modifies    \nothing;
  @   respects    anyUser, \result;
  @   declassify  ( \exists int i; 0<=i && i<names.length;
  @                 names[i]==user && passwords[i]==password
  @               ) \to \result;                                @*/
public boolean check(int user, int password) { ...
```

Fig. 6. Complete non-interference specification for the method `check` from Figure 2 with a seamless integration to the functional specifications.

In the next section it will be shown how the introduced information flow extensions of JML* can be translated to JavaDL and checked by the KeY-System.

3.4 Translating JML* Non-Interference Specifications to JavaDL

Summarising [27], the JML* extensions can be translated to JavaDL as follows. Views translate canonically to JavaDL terms which evaluate to a set of reference expressions in a given state. `respects`-clauses and declassifications can be translated to JavaDL by the formalisation of Definition 9:

Lemma 2. *Let $\mathscr{V}$ be a set of views over method m and let $\mathscr{D}$ be a set of declassifications. If the formula*

$$\begin{aligned}
&\forall Heap\, s^1_{in}, s^2_{in}, s^1_{out}, s^2_{out} \\
&\forall C\, this\, \forall \mathscr{T}_1\, p^1_1\ \ldots\ \forall \mathscr{T}_n\, p^1_n\, \forall \mathscr{T}_r\, r^1\, \forall \mathscr{T}_1\, p^2_1\ \ldots\ \forall \mathscr{T}_n\, p^2_n\, \forall \mathscr{T}_r\, r^2\, \Big(\\
&\qquad\qquad this \neq null \\
&\qquad\quad \wedge\, \mathscr{U}^1_{in} \langle \mathscr{T}_r\ result = this.m(p_1, \ldots, p_n) \rangle (h^1_{out} = heap \wedge r^1 = result) \\
&\qquad\quad \wedge\, \mathscr{U}^2_{in} \langle \mathscr{T}_r\ result = this.m(p_1, \ldots, p_n) \rangle (h^2_{out} = heap \wedge r^2 = result) \\
&\qquad\quad \rightarrow \bigwedge_{V \in \mathscr{V}} \big(\mathscr{U}^1_{in}\, V = \mathscr{U}^2_{in}\, V \wedge \bigwedge_{e \in \mathscr{U}^1_{in} V} \mathscr{U}^1_{in}\, e = \mathscr{U}^2_{in}\, e \wedge dcls \big) \\
&\qquad\qquad\qquad\qquad\qquad \rightarrow \mathscr{U}^1_{out}\, V = \mathscr{U}^2_{out}\, V \wedge \bigwedge_{e \in \mathscr{U}^1_{out} V} \mathscr{U}^1_{out}\, e = \mathscr{U}^2_{out}\, e \Big)
\end{aligned} \tag{2}$$

with updates $\mathcal{U}_x^i$ defined as $\mathcal{U}_x^i = \{heap := s_x^i \,||\, p_1 := p_1^i \,||\, \dots \,||\, p_n := p_n^i \,||\, r := r^i\}$ and with declassifications dlcs defined as

$$dcls := \bigwedge_{D \in \mathcal{D}} \left(\begin{array}{l} \quad \mathcal{U}_{in}^1\, D_{if} = true \wedge \mathcal{U}_{in}^2\, D_{if} = true \\ \wedge\, \mathcal{U}_{in}^1\, (V \subseteq D_{to}) \wedge \mathcal{U}_{in}^2\, (V \subseteq D_{to}) \\ \wedge\, D_{term} = \{heap := anon(heap, allLocs \setminus D_{from}, anonHeap\} D_{term} \\ \rightarrow \mathcal{U}_{in}^1\, D_{term} = \mathcal{U}_{in}^2\, D_{term} \end{array} \right) \tag{3}$$

is valid
then only declassified information flows from higher references to lower ones in m.

Proof. See [27].

4 Connection between Noninterference in KeY and Noninterference in an Open System

In this section, we argue that KeY can be used to prove noninterference in open systems, as specified in Definition 4 on page 11.

KeY can verify programs modularly on the level of individual methods. For this purpose method-contracts need to be specified for each method. More precisely, within the verification of a method *m* KeY can use contracts of other methods instead of unfolding their implementation.[5] A method *m* is verified if it has been proven that the implementation of *m* adheres to its contract(s). As shown in [3], not only functional method contracts can be used in this way but also *information flow contracts* [27], as defined in Section 3.3. In case a method does not have a known implementation (for instance if it is a native method or, as in our case study, if it is a part of the environment), a contract can be seen as an assumption on the behavior of the method. This assumption has to be justified externally.

By the results of Section 2.6, to show noninterference in an open system (Definition 4), it is enough to consider environments with only two (unspecified) static methods: `public static int untrustedInput()` and `public static void untrusted-Ouput(int x)`, as specified by the interface I_{IO} (see Section 2.6).

In order to show that the noninterference property proven by KeY (Lemma 2) implies noninterference in a system connected to such an environment (i.e. to an environment implementing I_{IO}), it has to be shown that

[5] Moreover, due to the concept of dynamic method dispatch in object-oriented languages such as JAVA, there may be several implementations. Since this dispatch is dynamic, it cannot be statically determined which thereof is the most precise applicable. One goal of KeY's calculus is to be modularily sound, i.e., it is sound with respect to open programs (see, e.g. [24]). One important property is that proofs are still correct if the program is changed in such ways that classes (and implementations of previously present methods) are added. This means that even in case where the static type of an instance is known, soundness requires to apply a contract because otherwise the addition of applicable method implementations would invalidate previous proofs.

```
1  public interface Environment {
2
3      /*@ public static model \locset envLocs;
4        @ accessible envLocs : envLocs;
5        @*/
6
7      /*@ normal_behavior
8        @     ensures     \new_elems_fresh(envLocs);
9        @     assignable  envLocs;
10       @     respects    envLocs, x;
11       @*/
12     public static void untrustedOuput(int x) {
13         // underspecified
14     }
15
16     /*@ normal_behavior
17       @     ensures     \new_elems_fresh(envLocs);
18       @     assignable  envLocs;
19       @     respects    envLocs;
20       @*/
21     public static int untrustedInput() {
22         // underspecified
23     }
24 }
```

Fig. 7. Specification of the environment interface.

1. the specification for the main method verified by KeY specifies the result value of E to be low, but the input-vector $\overrightarrow{x}$ of P as high and
2. the contracts of the methods in the interface I_{IO} of E are correctly chosen.

In a sense `envLocs` represents the complete knowledge of the environment. It will be defined as the set of low locations later on. The restricted form of this interface implies that P and E do not share any references: their sets of heap locations are disjoint. This disjointness will play an important part in the proofs. As a preview of things to happen we also point out that this disjointness property will be derived from the contracts in Figure 7.

Item (1) is obvious.

Figure 7 shows I_{IO} as a JAVA interface with an appropriate JML specification. In the following we will argue in detail that this JML specifications are correct, by which we mean that they are either justified by the JAVA semantics or by [13, Definition 18].

Line 3 defines a static model field `envLocs`. Intuitively the model field represents the heap locations belonging to the (unknown) environment. Formally the model field defines a function $envLocs : Heap \to LocSet$ which returns for any heap a set of (created) heap locations. The function is underspecified, but possible interpretations are restricted by line 4: let h be an arbitrary heap and let h' be a heap which differs from h only in locations not belonging to $envLocs(h)$. Then $envLocs(h) = envLocs(h')$ holds. Here we assume that *envLocs* is *self-framing*: the set of locations belonging to E cannot change if some value changes that does not belong to E. This assumption is justified by the fact that P and E interact only through the special interface I_{IO}: since I_{IO} does not allow the exchange of references, the set of locations belonging to E can only change if E creates new objects and stores references on these new objects or if E releases all references on an (own) object.

It remains to explain the contracts of `untrustedInput` and `untrustedOuput`. Let h_1 be the heap before an invocation of the method `untrustedOuput` and let h_2 be the heap after its invocation. The contract of `untrustedOuput` states the following:

– If values of locations in $envLocs(h_1)$ change (which means that $envLocs(h_1) \neq envLocs(h_2)$ might hold), then all locations in $envLocs(h_2) \setminus envLocs(h_1)$ are newly created locations (line 8). In other words: E cannot get references to heap locations not belonging to E. As before, the assumption is valid, because the parameter of `untrustedOuput` has a primitive type and E cannot callback P.
– `untrustedOuput` changes at most values of locations in $envLocs(h_1)$ or in newly created locations (line 9). Formally this is expressed by:

$$\forall l \notin envLocs(h_1) : created(h_I, l) \rightarrow select(h_1, l) = select(h_2, l)$$

Thus we assume that only values of locations currently belonging to E are changed by `untrustedOuput`. The assumption is valid for the same reasons as before.
– The behavior of `untrustedOuput` depends at most on the values of locations in $envLocs(h_1)$ and the value of the parameter `x` (line 10). This reflects the assumption that the environment cannot read from heap locations not belonging to the environment. The assumption is valid for the same reasons as before.

The contract of `untrustedInput` states the same as the contract of `untrustedOuput` except that the behavior of `untrustedInput` depends solely on the values of locations in $envLocs(h_1)$ (it has no parameter). The return value of `untrustedInput` is underspecified.

So far we have justified the assumptions on I_{IO} with the help of its special form, the requirement that P and E interact only through I_{IO} and the JAVA semantics. Finally we will consider the specification of the main method of P. Such a specification is given in Figure 8. Let h_1 be the heap before an invocation of `main` and let h_2 be the heap after its invocation. The vector $\overrightarrow{x}$ is represented in the example without loss of generality by the three attributes `x1`, `x2` and `x3`. An arbitrary number of `x`'es can be used instead. Furthermore P will use only locations of `\locset(x1, x2, x3)` and newly created locations.

The contract requires (line 5-6) that the locations belonging to `x1`, `x2` and `x3` are disjoint from $envLocs(h_1)$. Thus we require that the environment does not know the values of the `x`'es before the execution of `main`. Because P uses only locations of `\locset(x1, x2, x3)` and newly created locations, KeY is able to show with the help of the specification of I_{IO} that the locations of P and E (the latter described by *envLocs*) are always disjoint. Line 7 specifies the locations of $envLocs(h_1)$ as low in h_1 (before the execution of `main`). Furthermore line 7 specifies the locations of $envLocs(h_2)$ as low in h_2 (after the execution of `main`). As explained in Section 3.3, in our JML* extension everything

```
public class P {
    // arbitrary number of x'es
    private static int x1, x2, x3;

    /*@ normal_behavior
      @     requires    \disjoint(\locset(x1, x2, x3),
      @                           Environment.envLocs);
      @     respects    Environment.envLocs;
      @*/
    public static void main(String[] args) {
        // some secure program
    }

}
```

Fig. 8. Specification of the main method.

which is not explicitly defined as low is implicitly defined as high. Thus the locations of the x'es are implicitly defined as high in h_1.

Altogether KeY proves that for any environment E which implements I_{IO} and does not interact with P in any other way than through I_{IO}, for any interpretation of the function *envLocs* which adheres to the specified assumptions the values of the locations in $envLocs(h_2)$ depend at most on the values of the locations in $envLocs(h_1)$ (and in particular not on the values of the x'es in h_1). This proposition holds especially for environments E with a special field `result` and for interpretations of *envLocs* for which $\backslash locset(result) \subseteq envLocs(h_1)$ and $\backslash locset(result) \subseteq envLocs(h_2)$ holds. Hence we can conclude from a noninterference proof of KeY that noninterference in an open system holds.

5 The Case Study

In our case study, we consider a simple system that uses public-key encryption: clients send secrets encrypted over an untrusted network, controlled by an active adversary, to a server who decrypts the messages. This can be seen as a rudimentary way encryption can be used. Based on our framework, we started using the KeY tool to verify strong secrecy of the messages sent over the network, i.e., noninterference shall be shown using KeY for the system when it runs with IdealPKE and by our framework we then obtain computational indistinguishability guarantees when IdealPKE is replaced by RealPKE.

Clearly, the system itself is quite trivial from a cryptographic point of view. However, the point of language-based analysis in general and the point of our case study in particular is to show that the system is *implemented* in a way that the expected security guarantees actually hold true (there are more than enough opportunities to make implementation errors in even simple systems).

We emphasize that while the code of client and server are quite small, the actual code that needs to be analyzed is longer because it includes the ideal functionality and the code that results from applying the techniques of our general framework developed in Section 2 (we note the verified program is in the family of systems considered in this section); altogether the code, which in our case study comprises 10 classes, one interface, and about 30 methods and, in about 370 LoC of a rich fragment of JAVA.

Moreover, the adversary model we consider in the case study is strong in that the (active) adversary dictates the number of clients, sends a pair of messages to every client of which one is encrypted (in the style of a left-right oracle), and controls the network.

5.1 The Analyzed Program

We now describe the analyzed program in more detail. The code of the client and the server is given below.

```java
final public class Client {
  private Encryptor BobPKE;
  private byte[] message;

  public Client(Encryptor BobPKE, byte message) {
    this.BobPKE = BobPKE;
    this.message = new byte[] {message};
  }
  public void onInit() throws NetworkError {
    byte[] encMessage = BobPKE.encrypt(message);
    Network.networkOut(encMessage);
  }
}

final public class Server {
  private Decryptor BobPKE;
  private byte[] receivedMessage = null;

  public Server(Decryptor BobPKE) {
    this.BobPKE = BobPKE;
  }

  public void onReceive(byte[] message) {
    receivedMessage = BobPKE.decrypt(message);
  }
}
```

Besides the code for client and server, the program also contains a setup class which contains the methods `main()` and creates instances of protocol participants and organizes the communication. This setup first creates a public/private key pair (encapsulated in a decryptor object) for the server. In a `while`-loop it then expects, in every round, i) two input messages from the network (adversary), ii) depending on a static boolean variable `secret` (which will be declared to be *high*), one of the two messages is picked, iii) a client is created and it is given the public key of the server and the chosen message (the client will encrypt that message and send it over the network to the server), iv) a message from the network is expected, and v) given to the server, who will then decrypt this message and assign the plaintext to some variable.

We denote the class setup by Setup$[b]$, where $b \in \{$false, true$\}$ is the value with which secret is initialized in Setup. By $S^{real}[b]$, for $b \in \{$false, true$\}$, we denote the system consisting of the class Setup$[b]$, the class Client, the class Server, and the system RealPKE. This system is open: it uses unspecified network (and untrusted input from the environment) which is controlled by the adversary. Analogously, $S^{ideal}[b]$ contains IdealPKE instead of RealPKE. Note that $S^{ideal}[b]$ is even more open in that the ideal functionality asks the environment to encrypt and decrypt some messages (see the definition of IdealPKE).

The code for the setup class is as follows.

```java
public class Setup {
  static private boolean secret = b; // b ∈ {true, false}

  public static void main() throws NetworkError {
    // Public-key encryption functionality for Server
    Decryptor serverDec = new Decryptor();
    Encryptor serverEnc = serverDec.getPublicInterface();
    Network.networkOut(serverEnc.getPublicKey());

    // Creating the server
    Server server = new Server(serverDec);

    // The adversary decides how many clients we create
    while( Network.networkIn() != null ) {
      byte s1 = Network.networkIn()[0];
      byte s2 = Network.networkIn()[0];
      // and one of them is picked depending
      // on the value of the secret bit
      byte s = secret ? s1 : s2;
      Client client = new Client(serverEnc, s);
      // trigger the client
      client.onInit();
      // read a message from the network...
      byte[] message = Network.networkIn();
      // ... and deliver it to the server
      server.onReceive(message);
    }
  }
}
```

5.2 The Property to be Proven

The property we want to show is

$$S^{real}[\mathtt{false}] \approx^{\emptyset}_{\mathsf{comp}} S^{real}[\mathtt{true}], \tag{4}$$

that is, the two variants of the system are indistinguishable from the point of view of an adversary who implements the networking, but does not call (directly) methods of $S^{real}[b]$; he, however, through the setup class, determines the number of clients that are created and the message pair for every client.

By our framework, to prove (4) it is enough to show I-noninterference of the system $S^{ideal}[b]$. Since the system $S^{ideal}[b]$ is in the class of systems considered in Section 2.6, we can use the results from this section which say that we only need to show I-noninterference of the system $T[b] = E^*_{I_E} \cdot S^{ideal}[b]$ that uses much simpler interface I_{IO}.

The carried out verification shall establish (4), under the (reasonable) assumption that KeY is sound with respect to the subset of JAVA covered in Jinja+.

6 The Verification Process

The verification process in KeY consists of two parts: the specification of the program in JML and the proof that the program adheres to the specification.

The specification of the main method as well as the specification of the interface `Environment` have been discussed in great detail in Section 4. In our case study the `x`'s from Section 4 consist solely of the field `secret` of the class `Setup`. The rest of the necessary specification is related to modular verification: the size of the program is already that big that KeY needs to verify each method on its own. Hence each method has to be annotated with an appropriate contract. The contracts normally consist of two parts: the first part expresses heap separation properties which are preserved by the implementation. On the one hand these specifications are necessary for the modular verification, on the other hand they are necessary to prove that the locations of the environment are disjoint from the ones of the program. The second part of the specifications are information flow contracts. These are also needed for modularity reasons: during the verification of a method m we use the information flow contracts of the called methods to conclude that m has no flow. In this way we avoid unfolding the body of the called methods. Finally, the contracts might compromise preconditions. Most of the preconditions require that some invariants hold. Those invariants represent in large part the remaining specifications. In total the current specification has about 370 lines.

The specifications result in about 60 proof obligations. Currently we have proven 50 out of them. During the verification we realised two problems with our current approach.

The first problem is scalability. The key technique in KeY for modular (functional) verification, called *dynamic frames* [28], is quite new to KeY. So far it has mainly been tested on small examples. This case study is the first bigger example. It revealed that the automatic proof search strategy fails in finding proofs for examples with complex heap separation properties (including list specifications). In those cases the prover has to be guided by the user. This leads to semi-automatic proofs. This alone is not that much a problem, but because the heap separation properties are complex, it is also quite error-prone to specify those properties. Hence one normally needs several verification attempts until the right specification is found. Altogether this results in a very high verification effort. Until now we have spent about one to two person months on the case study. We will have to address this issue in future research in order to reduce the overall effort.

The second problem is related to the definition of non-interference in an object-oriented setting. It revealed that the standard definition of low-equivalence as used in this report is usually too restrictive for modular verification in an object-oriented setting. Already the simple method `m` in

```
class C {
  C low;
  m(){ low = new C(); }
}
```

(where low is a low variable) cannot be verified in a modular fashion. The reason is that we simply cannot guarantee that the creation of *the same* object in two heaps which agree on the low values, but nevertheless might contain a different number of created objects, will lead to the same heap location. Therefore, in order to carry out the security analysis, a richer notion of non-interference is needed in this setting.

This has lead us to the development of a notation of low-equivalence which takes object references into account [3]. Instead through the equality of all values, this definition relies on isomorphisms between heap structures. It is based on the assumption that an attacker is able to compare object references to each other (through ==), but not to infer other properties like the order or even the time of object creation.[6]

7 Conclusion

The case study supports the view that the general framework from [13] is a useful concept to extend the range of verification tasks that can be performed by deductive verification systems. The case study also revealed challenges for the verification with the KeY system. We are confident, and have first ideas on the measures to be taken, that these problems can be solved,

References

1. M. Alba-Castro, M. Alpuente, and S. Escobar. Abstract certification of global non-interference in rewriting logic. In F. S. de Boer, M. M. Bonsangue, S. Hallerstede, and M. Leuschel, editors, *Formal Methods for Components and Objects - 8th International Symposium (FMCO 2009). Revised Selected Papers*, volume 6286 of *Lecture Notes in Computer Science*, pages 105–124. Springer, 2009.
2. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. *Computer Security Foundations Workshop, IEEE*, 0:100, 2004.
3. B. Beckert, D. Bruns, V. Klebanov, C. Scheben, P. H. Schmitt, and M. Ulbrich. Secure information flow for Java. A Dynamic Logic approach. Technical report. available at http://i12www.ira.uka.de/ key/doc/2012/FMTR2012.pdf, KIT, 2012.
4. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach.* LNCS 4334. Springer, 2007.

[6] This would be possible in a language like C where pointers have integer values.

5. R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. Technical Report 2000/067, Cryptology ePrint Archive, December 2005. http://eprint.iacr.org/2000/067/.
6. Á. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Security in Pervasive Computing*, volume 3450 of *Lecture Notes in Computer Science*, pages 193–209. Springer Berlin / Heidelberg, 2005. 10.1007/978-3-540-32004-3_20.
7. J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
8. C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, December 2009.
9. D. Harel. *First-Order Dynamic Logic*, volume 68 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
10. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic*. The MIT Press, 2000.
11. S. Hunt and D. Sands. On flow-sensitive security types. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 79–90, New York, NY, USA, 2006. ACM.
12. G. Klein and T. Nipkow. A Machine-Checked Model for a Java-Like Language, Virtual Machine, and Compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006.
13. R. Kuesters, T. Truderung, and J. Graf. A Framework for the Cryptographic Verification of Java-like Programs. Cryptology ePrint Archive, Report 2012/153, 2012. http://eprint.iacr.org/2012/153.
14. R. Küsters. Simulation-Based Security with Inexhaustible Interactive Turing Machines. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW-19 2006)*, pages 309–320. IEEE Computer Society, 2006.
15. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31:1–38, May 2006.
16. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, and D. M. Zimmerman. *JML Reference Manual (Draft)*, July 2011. online available at http://www.eecs.ucf.edu/ leavens/JML/documentation.shtml.
17. B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, Oct. 1992.
18. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.
19. A. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 228–241. ACM, 1999.
20. A. C. Myers. JFlow: Practical mostly-static information flow control. In *POPL*, pages 228–241, 1999.
21. A. C. Myers, S. Chong, N. Nystrom, L. Zheng, and S. Zdancewic. *Jif: Java Information Flow (software release)*, July 2001. http://www.cs.cornell.edu/jif/.
22. B. Pfitzmann and M. Waidner. A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission. In *IEEE Symposium on Security and Privacy*, pages 184–201. IEEE Computer Society Press, 2001.
23. S. Ranise and C. Tinelli. The SMT-LIB standard, v 1.2. Technical report, U. of Iowa, 2006.
24. A. Roth. *Specification and Verification of Object-oriented Software Components*. PhD thesis, Universität Karlsruhe, 2006.
25. A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications, special issue on Formal Methods for Security*, 21(1):5–19, 2003.
26. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *J. Comput. Secur.*, 17:517–548, October 2009.

27. C. Scheben and P. H. Schmitt. Verification of information flow properties of Java programs without approximations. In *Formal Verification of Object-Oriented Software International Conference, FoVeOOS 2011, Revised Selected Papers*, LNCS. Springer, 2012. To appear. Earlier version in Technical Report 2011-26, KIT, Department of Informatics. Available at http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/1977984.
28. B. Weiß. *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction*. PhD thesis, Karlsruhe Institute of Technology, 2011.