2012 IEEE 25th Computer Security Foundations Symposium

# A Framework for the Cryptographic Verification of Java-like Programs

Ralf Küsters
*University of Trier, Germany*
*kuesters@uni-trier.de*

Tomasz Truderung
*University of Trier, Germany*
*truderung@uni-trier.de*

Jürgen Graf
*KIT, Germany*
*graf@kit.edu*

***Abstract*—We consider the problem of establishing cryptographic guarantees—in particular, computational indistinguishability—for Java or Java-like programs that use cryptography. For this purpose, we propose a general framework that enables existing program analysis tools that can check (standard) non-interference properties of Java programs to establish cryptographic security guarantees, even if the tools a priori cannot deal with cryptography. The approach that we take is new and combines techniques from program analysis and simulation-based security. Our framework is stated and proved for a Java-like language that comprises a rich fragment of Java. The general idea of our approach should, however, be applicable also to other practical programming languages.**

**As a proof of concept, we use an automatic program analysis tool for checking non-interference properties of Java programs, namely the tool Joana, in order to establish computational indistinguishability for a Java program that involves clients sending encrypted messages over a network, controlled by an active adversary, to a server.**

## I. Introduction

In this paper, we consider the problem of establishing security guarantees for Java or Java-like programs that use cryptography, such as encryption. More specifically, the security guarantees we are interested in are computational indistinguishability properties: Two systems $S_1$ and $S_2$ coded in Java, i.e., two collections of Java classes, are computationally indistinguishable if no probabilistic polynomially bounded environment (which is also coded as a Java program) is able to distinguish, with more than negligible probability, whether it interacts with $S_1$ or $S_2$. As a special case, $S_1$ and $S_2$ might only differ in certain values for certain variables. In this case, the computational indistinguishability of $S_1$ and $S_2$ means that the values of these variables are kept private, a property referred to as privacy, anonymity, or strong secrecy. Indistinguishability is a fundamental security property relevant in many security critical applications, such as secure message transmission, key exchange, anonymous communication, e-voting, etc.

**Our goal.** The main goal of this paper is to develop a general framework that allows us to establish cryptographic indistinguishability properties for Java programs using existing program analysis tools for analyzing (standard) non-interference properties [17] of Java programs, such as the tools Joana [19], KeY [1], a tool based on Maude [3], and Jif [27], [28]. As such, our work also contributes to the problem of implementation-level analysis of crypto-based software (such as cryptographic protocols) that has recently gained much attention (see Sections X and XI).

A fundamental problem that we face is that existing program analysis tools for non-interference properties cannot deal with cryptography directly. In particular, they typically do not deal with probabilities and the non-interference 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., [32] and references therein).

**Our approach.** Our approach to enabling these tools to nevertheless deal with cryptography and in the end provide cryptographic security guarantees is to use techniques from simulation-based security (see, e.g., [10], [31], [23]). The idea is to first analyze a (deterministic) Java program 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. As we show as part of our framework, we can then replace the ideal functionalities by their realizations, obtaining the actual Java program (without idealized components) with cryptographic guarantees.

**Our contribution in more detail.** More precisely, our approach and the contribution of this paper are as follows.

Our framework is formulated for a language we call Jinja+ and is proven w.r.t. the formal semantics of this language. Jinja+ is a Java-like language that extends the language Jinja [22] and comprises a rich fragment 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`.

Along the lines of simulation-based security, we formulate, in Jinja+, rather than in a Turing machine model, what it means for two systems to be computationally indistinguishable, and for one system to realize another system.


DOI 10.1109/CSF.2012.9

We then prove a composition theorem that allows us to replace ideal functionalities (formulated as Jinja+ systems) by their realizations (also formulated as Jinja+ systems) in a more complex system. The definitions and proofs need care because interfaces between different Jinja+ systems are classes with their fields and methods, and hence, these interfaces are very different and much richer than in the case of interactive Turing machines, where machines are simply connected by tapes on which bit strings are exchanged.

As mentioned before, we are mainly interested in establishing computational indistinguishability properties for crypto-based Java or Java-like programs (i.e., programs that use cryptography) using existing analysis tools for language-based information flow analysis. At the core of our approach, sketched above, is a theorem that says that if two systems that both use an ideal functionality are perfectly indistinguishable, then these systems are computationally indistinguishable if the ideal functionality is replaced by its realization, where perfect indistinguishability is defined (for deterministic Java programs) just as computational indistinguishability but w.r.t. unbounded adversaries. Together with another theorem that we obtain, and which states that (termination-insensitive) non-interference [32] is equivalent to perfect indistinguishability, we obtain that by proving non-interference using existing program analysis tools, we can establish computational indistinguishability properties.

Many program analysis tools can deal only with closed Java programs. The systems we want to analyze are, however, open, because they interact with a network or use some libraries that we do not (have to) trust, and hence, do not have to analyze. In our setting, the network and such libraries are simply considered to be part of the environment (the adversary). As part of our framework, we therefore also propose proof techniques that help program analysis tools to deal with these kinds of open systems, and in particular, to prove non-interference properties about these systems. These techniques are used in our case study (see below), but they are rather general, and hence, relevant beyond our case study.

Since we use public-key encryption in our case study, we also propose an ideal functionality for public-key encryption coded in Jinja+, in the spirit of similar functionalities in the simulation-based approach (see, e.g., [10], [25]), and prove that it can be realized with any IND-CCA2-secure public-key encryption scheme. This result is needed whenever a Java system is analyzed that uses public-key encryption, and hence, is relevant beyond our case study. We note that the formulation of our ideal functionality is more restricted than the one in the cryptographic literature in that corruption is not handled within the functionality.

As a case study and as a proof of concept of our framework and approach, we consider a simple Java program, which in fact falls into the Jinja+-fragment of Java and in which clients (whose number is determined by an active adversary) encrypt secrets under the public key of a server and send them, over an untrusted network controlled by the active adversary, to a server who decrypts these messages. Using the program analysis tool Joana [19], which is a fully automated tool for proving non-interference properties of Java programs, we show that our system enjoys the non-interference property (with the secrets stored in high variables) when the ideal functionality is used instead of real encryption. The theorems proved in our framework thus imply that this system enjoys the computational indistinguishability property (in this case strong secrecy of the secrets) when the ideal functionality is replaced by its realization, i.e., the actual IND-CCA2-secure public-key encryption scheme.

**Structure of the paper.** The language Jinja+ is introduced in Section II. Perfect and computational indistinguishability for Jinja+ systems are formulated in Section III. In Section IV, we define simulation-based security for Jinja+ and present the mentioned composition theorem. The relationship between computational and perfect indistinguishability as well as non-interference is shown in Sections V and VI. The proof technique for non-interference in open systems is discussed Section VII. The ideal functionality for public-key encryption and its realization can be found in Section VIII, with the case study presented in Section IX. In Section X, we discuss related work. We conclude in Section XI. More details and full proofs are provided in the extended version of this paper [24].

## II. JINJA+: A JAVA-LIKE LANGUAGE

As mentioned in the introduction, our framework is stated for a Java-like language which we call *Jinja+*. Jinja+ is based on *Jinja* [22] and extends this language with some additional features that are useful or needed in the context of our framework.

Jinja+ covers a rich subset of Java, including classes, inheritance, (static and non-static) fields and methods, the primitive types `int`, `boolean`, and `byte` (with the usual operators for these types), arrays, exceptions, and field/method access modifiers, such as `public`, `private`, and `protected`. Among the features of Java that are *not* covered by Jinja+ are: abstract classes, interfaces, strings, and concurrency. We believe that extending our framework to work for these features of Java, except for concurrency, is quite straightforward. We leave such extensions for future work.

We now first recall the Jinja language and then present the extended language Jinja+.

### A. Jinja

**Syntax.** 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 syntactically 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 syntactically correct (can be compiled), but possibly incomplete (can use not defined classes). In particular, a system can be extended to a (complete) program.

Some constructs 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. In this paper we consider only well-typed programs.

**Semantics.** Following [22], 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 [24].

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 $s'$.

### B. Jinja+

As a basis of our formal results we take a language that extends Jinja with: (a) the primitive type `byte` with

```
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
  }
}
```

Figure 1. An example Jinja+ program (in Java-like notation).

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 adopted in this paper 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 our 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 not make use of `randomBit()` are (called) *deterministic*, and otherwise, they are called *randomized*.

An example of a Jinja+ program is, as mentioned, presented in Figure 1.

**Runs of Jinja+ programs.** As already mentioned, the full set of rules of the small-step semantics of Jinja+ can be found in [24]. Based on this small-step semantics, we now define runs of Jinja+ programs.

**Definition 1.** A *run* of a deterministic program $P$ is a sequence of configurations obtained using the (small-step) Jinja+ semantics from the initial configuration of the form $\langle e_0, (h_0, l_0)\rangle$, where $e_0 = C.\texttt{main()}$, 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 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 $C$.

## III. Indistinguishability

We now define what it means for two systems to be indistinguishable by environments interacting with those systems. Indistinguishability is a fundamental relationship between systems which is interesting in its own right, for example, to define privacy properties, and to define simulation-based security, as we will see in the subsequent sections.

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 already mentioned in the introduction, 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.

Our definitions of indistinguishability follow the spirit of definitions of (computational) indistinguishability in the cryptographic literature (see, e.g., [10], [23], [20]), but, of course, instead of interactive Turing machines, we consider Jinja+ systems/programs. In particular, the simple communication model based on tapes is replaced by rich object-oriented interfaces between subsystems.

### A. Interfaces and Composition

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.

**Definition 2.** 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.

**Definition 3.** 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 $S : \emptyset$.

**Definition 4.** We say that *a system $S$ uses an interface $I$*, written $I \vdash S$, if, besides its own classes, $S$ 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.

**Definition 5.** 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:

**Definition 6 (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.

We emphasize that the interfaces between subsystems as considered here are quite different and much richer than the interfaces between interactive Turing machines considered in cryptography. Instead of plain bit strings that are sent over tapes between different machines, objects can be created, classes of another subsystem can be extended by inheritance, and data of different types, including references pointing to possibly complex objects, can be passed between different objects. Also, the flow of control is different. While in the Turing machine model, sending a message gives control to the receiver of the message and this control might not come back to the sender, in the object-oriented setting communication goes through method calls and fields. After a method call, control comes back to the caller, provided that potential exceptions are caught and the execution is not aborted.

We also emphasize that while a setting of the form $\vdash S : I$ and $I \vdash T$, i.e., in $S \cdot T$ the system $T$ uses the interface $I$ implemented by $S$, suggests that the initiative of accessing fields and calling methods always comes from $T$, it might also come from $S$ by using *callback objects*: $T$ could extend classes of $S$ by inheritance, create objects of these classes and pass references to these objects to $S$ (by calling methods of $S$). Then, via these references, $S$ can call methods specified in $T$. (This, in fact, is a common programming technique.)

### B. Environments

An environment will interact with one of two systems and it has to decide with which system it interacted (see Section III). Its decision is written to a distinct static boolean variable `result`.

**Definition 7.** 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.

**Definition 8.** 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 in the above definition, 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 $E \cdot S \rightsquigarrow$ `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`.

**Definition 9 (same interface).** 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 $S_2$.

### C. 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.

For a natural number $\eta$, we define a system $SP_\eta$ that implements the interface $I_{SP}$ by setting the initial value of `securityParameter` to $\eta$. We do not fix here how this value is represented because the representation is not essential for our results; it could be represented as a linked list of objects or an array (see also the discussion below).

We call a system $P$ such that $I_{SP} \vdash P$ a *program with a security parameter* or simply a *program* if the presence of a security parameter is clear from the context. Note that by this, $SP_\eta \cdot P$ is a complete program, which we abbreviate by $P(\eta)$.

Although as far as asymptotic security is concerned, our framework works fine with the definitions we have introduced so far, they are not perfectly aligned with the common practice of programming in Java. More specifically, messages, such as keys, ciphertexts, digital signatures, etc., are typically represented as arrays of bytes. However, this representation is bounded by the maximal length of an array, which is the maximal value of an integer (`int`). Therefore, following common programming practice, there would be a strict bound on, for example, the maximal size of keys (if represented as arrays of bytes). Since we consider asymptotic security, the size of keys should, however, grow with the security parameter.

One solution could be to use another representation of messages, such as lists of bytes. This, however, would result in unnatural programs and we, of course, want to be able to analyze programs as given in practice. Another solution could be to use concrete instead of asymptotic security definitions. However, most results in simulation-based security are formulated w.r.t. asymptotic security, and hence, we would not be able to reuse these results and avoid, for example, reproving from scratch realizations of ideal functionalities.

Therefore, we prefer the following solution. We parameterize the semantics of Jinja+ with the maximal (absolute) value integers can take. So if $P$ is a deterministic program, the *run of $P$ with integer size $s \geq 1$* is a run of $P$ where the maximal (absolute) value of integers is $s$; analogously for randomized programs. We write $P \rightsquigarrow_s \texttt{true}$ if the output of such a run is `true`; analogously, we define $\mathsf{Prob}\{P \rightsquigarrow_s \texttt{true}\}$. In our asymptotic formulations of indistinguishability, the size of integers may depend on the security parameter.

### D. Perfect Indistinguishability

We now formulate perfect indistinguishability, which, as we will prove in Section V, implies computational indistinguishability. We say that a deterministic program $P$ *terminates for integer size $s$*, if the run of $P$ with integer size $s$ is finite.

**Definition 10 (Perfect indistinguishability).** Let $S_1$ and $S_2$ be deterministic systems with a security parameter and 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, $S_2$) with security parameter, for every security parameter $\eta$ and every integer size $s \geq 1$, it holds that if $E \cdot S_1(\eta)$ and $E \cdot S_2(\eta)$ terminate for integer size $s$, then $E \cdot S_1(\eta) \rightsquigarrow_s \texttt{true}$ iff $E \cdot S_2(\eta) \rightsquigarrow_s \texttt{true}$.

We note that the notion of perfect indistinguishability introduced above is *termination-insensitive*, i.e. it puts no restrictions on non-terminating runs. This (weak) form of perfect indistinguishability, nevertheless implies computational indistinguishability (see Theorem 3).

### E. 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.

For this purpose, we fix now and for the rest of this section a polynomially computable function *intsize* that takes a security parameter $\eta$ as input and outputs a natural number $\geq 1$. We require that the numbers returned by this function are bounded by a fixed polynomial in the security parameter. All notions defined in what follows are parameterized by that function. However, due to ease of notion this will not be made explicit.

Our runtime definitions follow the spirit of definitions in cryptographic definitions of simulation-based security, in particular, [23], [20].

We start with the definition of almost bounded programs. These are programs that, with overwhelming probability, terminate after a polynomial number of steps.

**Definition 11 (Almost bounded).** 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 $P(\eta)$ (with integer size $\mathit{intsize}(\eta)$) exceeds $p(\eta)$ is a negligible function in $\eta$.[1]

It is easy to see that an almost bounded program $P$ can be simulated by a probabilistic polynomial time Turing machine that simulates at most $p(\eta)$ steps of a run of $P(\eta)$ (with integer size $\mathit{intsize}(\eta)$) and produces output that is distributed the same up to a negligible difference.

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.

**Definition 12 (Bounded environment).** 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 $E \cdot S(\eta)$ (with integer size $\mathit{intsize}(\eta)$), the number of steps performed in the code of $E$ does not exceed $p(\eta)$.

This definition makes sense since $E$ can abort a run by calling `abort()`, and hence, $E$ can prevent to be called by $S$, which would always require to execute some code in $E$. (Without `abort()`, $E$ can, in general, not prevent that code fragments of $E$ are executed, e.g., $S$ could keep calling methods of classes of $E$.)

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.

**Definition 13 (Environmentally $I$-bounded).** 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.

It is typically easy to see that a system is environmentally $I$-bounded (for all functions *intsize*).

We note that environmentally $I$-bounded systems, as defined above, are reactive systems that are free to process an unbounded number of requests of the environment $E$. In particular, a reactive system $S$ does not need to terminate after some fixed and bounded number of requests. Clearly, every bounded $I$-environment, being bounded, will only invoke $S$ a bounded number of times, more precisely, the number of invocations the environment makes is bounded by some polynomial in the security parameter.

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

### F. Computational Indistinguishability

Having defined polynomially bounded systems and programs, we are now ready to define computational indistinguishability of systems, where, again, we fix the function *intsize*. (However, computational guarantees for Java programs will be independent of a specific function *intsize*.) We start with the notion of computationally equivalent programs.

**Definition 14 (Computational Equivalence).** 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 $|\mathsf{Prob}\{P_1(\eta) \rightsquigarrow_{intsize(\eta)} \mathtt{true}\} - \mathsf{Prob}\{P_2(\eta) \rightsquigarrow_{intsize(\eta)} \mathtt{true}\}|$ is a negligible function in the security parameter $\eta$.

**Definition 15 (Computational indistinguishability).** 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 $E \cdot S_1 \equiv_{\mathsf{comp}} E \cdot S_2$.

Typically, this definition is applied to systems $S_1$ and $S_2$ that do not use the statement `abort()`. However, our results also work in this case.

We also note that this definition of indistinguishability is w.r.t. uniform environments. A definition w.r.t. non-uniform environments can be obtained in a straightforward way by giving the environment additional auxiliary input (besides the security parameter).

Furthermore, 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 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 (see Section III-A). (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.)

## IV. Simulatability

We now formulate what it means for a system to realize another system, in the spirit of the simulation-based approach.

As before, we fix a function *intsize* (see Section III-E) for the rest of this section. Typically, one would prove that one system realizes the other for all such functions.

Our formulation of the realization of one system by another follows the spirit of strong simulatability in the simulation-based approach (see, e.g., [23]). 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 16 (Strong Simulatability).** 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 $I_E$ from an (untrusted) environment, for example, networking and certain other libraries. In addition, the ideal functionality $F$ may require services $I_S$ from the simulator $S$, which in turn may require services $I_E$ from the environment. Recall from the discussion in Section III-A that the interfaces can be very rich—they allow for communication and method calls in both directions.

In the applications we envision, with our case study being the first example, $F$ will typically be an ideal functionality for one or more cryptographic primitives and its realization $R$ will basically be the actual cryptographic schemes.

The notion of strong simulatability, as introduced above, enjoys important basic properties, namely, reflexivity and transitivity, and allows to prove a fundamental composition theorem.

**Lemma 1 (Reflexivity of strong simulatability).** *Let $I_{out}$, $I_{in}$, and $I_E$ be disjoint interfaces and let $R$ be a system such that $I_E \cup I_{in} \vdash R : I_{out}$ and $R$ is environmentally $I_{out}$-bounded. Then, $R \leq R$, i.e., $R$ realizes itself.*

**Lemma 2 (Transitivity of strong simulatability).** *Let $I_{out}$, $I_{in}$, $I_E$, $I_S^0$, and $I_S^1$ be disjoint interfaces and let $R_0$, $R_1$, and $R_2$ be environmentally $I$-bounded systems. If $R_1 \leq^{(I_{out},I_{in},I_E \cup I_S^1,I_S^0)}$*

$R_0$ *and* $R_2 \leq^{(I_{out},I_{in},I_E,I_S^1)} R_1$*, then* $R_2 \leq^{(I_{out},I_{in},I_E,I_S^0 \cup I_S^1)} R_0$.

In short, the following composition theorem says that if $R_0$ realizes $F_0$ and $R_1$ realizes $F_1$, then the composed real system $R_0 \cdot R_1$ realizes the composed ideal system $F_0 \cdot F_1$. In other words, it suffices to prove the realizations separately in order to obtain security of the composed systems.

**Theorem 1 (Composition Theorem).** *Let* $I_0$*,* $I_1$*,* $I_E$*,* $I_S^0$*, and* $I_S^1$ *be disjoint interfaces and let* $R_0$*,* $F_0$*,* $R_1$*, and* $F_1$ *be systems such that* $R_0 \leq^{(I_0,I_1,I_E,I_S^0)} F_0$*,* $R_1 \leq^{(I_1,I_0,I_E,I_S^1)} F_1$*, not both* $R_0$ *and* $R_1$ *contain* `main`*(), and* $R_0 \cdot R_1$ *are environmentally* $(I_0 \cup I_1)$*-bounded. Then,* $R_0 \cdot R_1 \leq^{(I_0 \cup I_1, \emptyset, I_E, I_S^0 \cup I_S^1)} F_0 \cdot F_1$.

For simplicity, Theorem 1 is stated in such a way that the trusted service that $R_i$ may use is completely provided by $R_{i-1}$, namely through $I_{i-1}$. It is straightforward (only heavy in notation) to state and prove the theorem for the more general case that the trusted service is only partially provided by the other system.

## V. From Perfect to Computational Indistinguishability

We now prove 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. As already explained in the introduction, 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. As before, we fix a function *intsize* (see Section III-E) for the rest of this section.

The proof is done via two theorems. 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. To prove this theorem, we need the following lemma.

**Lemma 3.** *Let* $I_E$ *and* $I$ *be disjoint interfaces and let* $S_1$ *and* $S_2$ *be environmentally* $I$*-bounded systems such that* $S_1 \approx^I_{\mathsf{comp}} S_2$ *(in particular,* $S_1$ *and* $S_2$ *use the same interface) and* $I_E \vdash S_1 : I$*, and hence,* $I_E \vdash S_2 : I$*. Let* $E$ *be a (not necessarily bounded)* $I$*-environment for* $S_1$ *(and hence,* $S_2$*) with* $I \vdash E : I_E$ *such that* $E \cdot S_1$ *is almost bounded. Then* $E \cdot S_2$ *is almost bounded and* $E \cdot S_1 \equiv_{\mathsf{comp}} E \cdot S_2$.

**Theorem 2.** *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* `main`*() iff* $P_2$ *contains* `main`*(), (iv) not both* $P_1$ *and* $F$ *(and hence,* $R$*) contain* `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* $R \cdot P_1 \approx^J_{\mathsf{comp}} R \cdot P_2$.

*Proof:* Assume that $F \cdot P_1 \approx^J_{\mathsf{comp}} F \cdot P_2$. In particular, $F \cdot P_1$ and $F \cdot P_2$ use the same interface and, therefore $R \cdot P_1$ and $R \cdot P_2$ use the same interface as well.

Let $E$ be a bounded $J$-environment for $R \cdot P_1$. We need to show that $E \cdot R \cdot P_1 \equiv_{\mathsf{comp}} E \cdot R \cdot P_2$.

Because $R \leq^{(I,I_P,I_E,I_S)} F$, there exists a simulator $S$ such that $I_E \vdash S : I_S$, $S \cdot F$ is environmentally $I$-bounded and

$$R \approx^I_{\mathsf{comp}} S \cdot F \tag{1}$$

Now, because $R \cdot P_i$, $i \in \{1,2\}$, is environmentally $J$-bounded, the system $E \cdot R \cdot P_i$ is almost bounded. By (1) and Lemma 3 we can conclude that $E \cdot S \cdot F \cdot P_i$ is almost bounded and

$$E \cdot R \cdot P_i \equiv_{\mathsf{comp}} E \cdot S \cdot F \cdot P_i \ . \tag{2}$$

As we have assumed that $F \cdot P_1 \approx^J_{\mathsf{comp}} F \cdot P_2$, by Lemma 3 we obtain

$$E \cdot S \cdot F \cdot P_1 \equiv_{\mathsf{comp}} E \cdot S \cdot F \cdot P_2. \tag{3}$$

Combining (2) and (3), we obtain $E \cdot R \cdot P_1 \equiv_{\mathsf{comp}} E \cdot R \cdot P_2$. ■

For simplicity of presentation, the theorem is formulated in such a way that $P_i$, $i \in \{1,2\}$, uses only $I$ as a (trusted) service and $F/R$ uses $I_P$. It is straightforward to also allow for other external services.

We show that perfect indistinguishability implies computational indistinguishability.

**Theorem 3.** *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 2 and Theorem 3, we obtain the desired result explained at the beginning of this section.

**Corollary 1.** *Under the assumption of Theorem 2 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* $P_1 \cdot R \approx^J_{\mathsf{comp}} P_2 \cdot R$.

Recall that $P_1 \cdot R \approx^J_{\mathsf{comp}} P_2 \cdot R$ is (implicitly) defined w.r.t. the integer size function $\mathit{intsize}(\eta)$. However, since the statement $P_1 \cdot F \approx^J_{\mathsf{perf}} P_2 \cdot F$ does not depend on any integer size function, we obtain that computational indistinguishability holds independently of a specific integer size function.

## VI. Perfect Indistinguishability and Non-Interference

In this section we prove that perfect indistinguishability and non-interference are equivalent. Hence, in combination with Corollary 1 it suffices for tools to analyze systems that use an ideal functionality w.r.t. non-interference in order to get computational indistinguishability for systems when the ideal functionality is replaced by its realization. As already mentioned in the introduction, many tools can analyze non-interference for Java programs.

The (standard) non-interference notion for confidentiality [17] requires the absence of information flow from high to low variables within a program. In this paper, we define non-interference for a (Jinja+) program $P$ with some static variables $\vec{x}$ of primitive types that are labeled as high. Also, some other static variables of primitive types are labeled as low. We say that $P[\vec{x}]$ is a *program with high and low variables*. By $P[\vec{a}]$ we denote the program $P$ where the high variables $\vec{x}$ are initialized with values $\vec{a}$ and the low variables are initialized as specified in $P$. We assume that the length of $\vec{x}$ and $\vec{a}$ are the same and $\vec{a}$ contains values of appropriate types; in such a case we say that $\vec{a}$ is valid. Now, non-interference for a (deterministic) program is defined as follows.

**Definition 17** (**Non-interference for Jinja+ programs).** Let $P[\vec{x}]$ be a program with high and low variables. Then, $P[\vec{x}]$ *has the non-interference property* if the following holds: for all valid $\vec{a}_1$ and $\vec{a}_2$ and all integer sizes $s \geq 1$, if $P[\vec{a}_1]$ and $P[\vec{a}_2]$ terminate for integer size $s$, then at the end of these runs, the values of the low variables are the same.

Similarly to the definition of perfect indistinguishability (Definition 10), the above definition captures *termination-insensitive* non-interference.

We note that the non-interference property is quite powerful: $P$ could have just one high variable of type boolean. Depending on the value of this variable $P$ could run one of two systems $S_1$ and $S_2$, illustrating that the non-interference property can be as powerful as perfect indistinguishability.

The above notion of non-interference deals with complete programs (closed systems). We now generalize this definition to open systems:

**Definition 18** (**Non-interference in an open system).** Let $I$ be an interface and let $S[\vec{x}]$ be a (not necessarily closed) deterministic system with a security parameter, high and low variables, and such that $S : I$. Then, $S[\vec{x}]$ is *$I$-noninterferent* if for every deterministic $I$-environment $E$ for $S[\vec{x}]$ and every security parameter $\eta$ non-interference holds for the system $E \cdot S[\vec{x}](\eta)$, 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 $I$-non-interference and perfect indistinguishability:

**Theorem 4.** *Let $I$ and $S[\vec{x}]$ be given as in Definition 18 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 $\vec{a}_1$ and $\vec{a}_2$, we have that $S[\vec{a}_1] \approx^I_{\mathsf{perf}} S[\vec{a}_2]$.*
*(b) $I$-non-interference holds for $S[\vec{x}]$.*

As already explained in the introduction and at the beginning of this section, in combination with Corollary 1, this theorem reduces the problem of checking computational indistinguishability for systems that use real cryptographic schemes to checking non-interference for systems that use ideal functionalities.

## VII. A Proof Technique for Non-interference in Open Systems

There are many tools that can deal with classical non-interference for closed systems, i.e., complete programs. In this section, we develop proof techniques that enable such tools, at least some of them, to also deal with open systems. Technically, we will show that in order to check non-interference for open systems according to Definition 18 it is sufficient to consider only a very restricted class of environments, rather than all environments. The input these environments give to the system they interact with is fixed for every environment and does not depend on the output the environment got from the system. In fact, the environments in this class are all almost identical, they only differ in the input sequence that they use. Now, the analysis a tool performs often ignores or can ignore specific values encoded in the program, such as the input sequence. So, if such an analysis establishes non-interference for a system and a fixed environment in the considered class, then this implies that non-interference holds for all environments in this class. By our proof technique, it then follows that non-interference holds for all environments, as required by Definition 18.

In our case study, we demonstrate that using our proof techniques, the tool Joana, which is designed to check non-interference for closed systems, can now deal with (a relevant class of) open systems as well.

To illustrate the main idea, we start with a simple case where communication between the environment and the system is restricted to primitive types only. We then generalize the result to the case where the environment and the system can communicate using also exceptions, arrays of bytes, and simple objects.

### A. Communication through Primitive Types Only

In this section, we assume that a system $S$ communicates with an environment $E$ only through static functions with primitive types. More precisely, we consider programs $S$ such that (1) method `main` is defined in $S$ and (2) $I_E \vdash S$, for some interface $I_E$, where all methods are static, use primitive types only (for simplicity of presentation we will consider only the type `int`), and have empty `throws` clause. We will consider indistinguishability w.r.t. the empty interface (i.e. environments we consider do not directly call $S$).

The above assumptions will allow us to show, in the proof of Theorem 5, that $E$ and $S$ do not share any references: their states are in this sense disjoint.

We now define the class of environments mentioned at the beginning of this section. We then show that to establish $I$-non-interference, it is enough to consider only those environments (see Theorem 5).

```
1  class Node {
2      int value;
3      Node next;
4      Node(int v, Node n) { value = v;  next = n; }
5  }
6  private static Node list = null;
7  private static boolean listInitialized = false;
8  private static Node initialValue()
9      { return new Node(u1, new Node(u2, ...)); }
10 static public int untrustedInput() {
11     if (!listInitialized)
12         { list = initialValue(); listInitialized = true; }
13     if (list==null) return 0;
14     int tmp = list.value;
15     list = list.next;
16     return tmp;
17 }
18 static public void untrustedOutput(int x) {
19     if (untrustedInput()!=0) {
20         result = (x==untrustedInput());
21         abort();
22     }
23 }
```

Figure 2. Implementation of `untrustedInput` and `untrustedOutput` in $\tilde{E}_{\vec{u}}$. We assume that class `Node` is not used anywhere else.

For a finite sequence $\vec{u} = u_1, \ldots, u_n$ of values of type `int`, we denote by $\tilde{E}_{\vec{u}}^{I_E}$ the following system.

The environment $\tilde{E}_{\vec{u}}^{I_E}$ contains two static methods: `untrustedOutput` and `untrustedInput`, as specified in Figure 2. For the sake of the discussion, let $S$ be the system the environment $\tilde{E}_{\vec{u}}^{I_E}$ interacts with. (Note that the definition of $\tilde{E}_{\vec{u}}^{I_E}$ is independent of a specific $S$. It only depends on $I_E$.) As we will see below, the method `untrustedOutput` gets all data passed by $S$ to $\tilde{E}_{\vec{u}}^{I_E}$. The method `untrustedInput` determines what the environment passes on to $S$.

More specifically. the method `untrustedInput()` returns consecutive values of $\vec{u}$ and, after the last element of $\vec{u}$ has been returned, it returns 0. Note that the consecutive values returned by this method are hardwired in line 9 (determined by $\vec{u}$) and do not depend on any input to $\tilde{E}_{\vec{u}}^{I_E}$.

The method `untrustedOutput`, depending on the values given by `untrustedInput()`, either ignores its argument or compares its value to the next integer it obtains, again, from `untrustedInput()` and stores the result of this comparison in the (low) variable `result`. The intuition is the following: `untrustedOutput` gets, as we will see below, all the data the environment gets from $S$. If the two instances of $S$, $S[\vec{a}_1]$ and $S[\vec{a}_2]$, which the environment tries to distinguish, behave differently, then there must be some point where the environment gets different data from the two systems in the corresponding runs, i.e., `untrustedOutput` will be invoked with different values for `x`, say the values `x` takes at this point are $b_1$ and $b_2$, respectively. By choosing an appropriate $\vec{u}$, this can be detected by `untrustedOutput`: $\vec{u}$ should be defined in such a way that the method `untrustedInput()` returns 1 at this point and that the value `untrustedInput()`

```
24 static public int foo(int x) {
25     untrustedOutput(FOO_ID);
26     untrustedOutput(x);
27     return untrustedInput()
28 }
```

Figure 3. $\tilde{E}_{\vec{u}}^{I_E}$: the implementation of a method of $I_E$ with the signature `static public int foo(int x)`, where `FOO_ID` is an integer constant serving as the identifier of this method (we assign a different identifier to every method).

returns next equals $b_1$, say ($b_2$ would also work). Then, in the run of the environment with $S[\vec{a}_1]$ the variable `result` will be assigned 1 and in the run with $S[\vec{a}_2]$ it will be assigned 0. Hence, the environment successfully distinguished $S[\vec{a}_1]$ and $S[\vec{a}_2]$.

Finally, for every method declaration $m$ in $I_E$, the system $\tilde{E}_{\vec{u}}^{I_E}$ contains the implementation of $m$ as illustrated by the example in Figure 3. As we can see, the defined method forwards all its input data to `untrustedOutput` and lets `untrustedInput` determine the returned value.

This completes the definition of $\tilde{E}_{\vec{u}}^{I_E}$. The next theorem states that, to prove $I$-non-interference, it is enough to consider only environments $\tilde{E}_{\vec{u}}^{I_E}$ for all $\vec{u}$. In this way we need to study only (almost) closed systems, namely systems that differ in only one expression (line 9). As discussed at the beginning of this section, this restriction is often sufficient for tools are designed to deal with closed systems only.

**Theorem 5.** *Let $I_E$ be an interfaces with only static methods of primitive argument and return types as introduced above. Let $S$ be a system with high and low variables such that `main` is defined in $S$ and $I_E \vdash S$. Then, $I$-non-interference, for $I = \emptyset$, holds for $S$ if and only if for all sequences $\vec{u}$ as above non-interference holds for $\tilde{E}_u^{I_E} \cdot S$.*

### *B. Communication through Arrays, Simple Objects, and Exceptions*

The result stated in Theorem 5 can be extended to cover some cases where, $E$ and $S$ exchange information not only through values of primitive types, but also through arrays, simple object (i.e. objects containing only fields of primitive types and arrays), and through throwing exceptions. Some restrictions, however, have to be imposed on $I_E$ and the program $S$. These restrictions, formally specified in [24], guarantee that, although references are exchanged between $E$ and $S$, the communication resembles exchange of pure data.

## VIII. Public-key Encryption

In our case study, we will analyze a system that uses public-key encryption. We therefore now provide an ideal functionality for public-key encryption, denoted by IdealPKE. Our example program will be analyzed based on this functionality (see Section IX). We prove that this

functionality can be realized by a system, which we call RealPKE, that implements, in the obvious way, an IND-CCA2-secure public-key encryption scheme.

The interface implemented by IdealPKE and RealPKE, denoted by $I_{PKE}$, consists of two classes:

```
public final class Decryptor {
    public Decryptor(); // class constructor
    public Encryptor getEncryptor();
    public byte[] decrypt(byte[] message);
}
public final class Encryptor {
    public byte[] getPublicKey();
    public byte[] encrypt(byte[] message);
}
```

An object of class `Decryptor` is supposed to encapsulate a public/private key pair. These keys are created when the object is constructed. It allows a party who owns such an object to decrypt messages (for which, intuitively, the private key is needed) by the method `decrypt`. This party can also obtain an encryptor which encapsulates the related public key and encrypts messages via the method `encrypt`. The encapsulated public key can be obtained by the method `getPublicKey`. Typically, the party who creates (owns) a decryptor gives away only an associated encryptor.

Our ideal functionality IdealPKE is in the spirit of ideal public-key functionalities in the cryptographic literature (see, e.g., [10], [25]). However, it is more restricted in that we do not (yet) model corruption in that functionality, and hence, in its realization. So far, corruption, if considered, needs to be modeled in the higher-level system using IdealPKE. In particular, whenever an encryptor object is used, it is guaranteed that the public key encapsulated in this object was honestly generated and no party has direct access to the corresponding private key.

While this might be too restricted and inconvenient in some scenarios (and it is interesting future work to extend this functionality), we took this design choice because our functionality facilitates the automated verification process and is nevertheless useful in interesting scenarios. We emphasize that the theorem for the realization of IdealPKE (Theorem 6) would hold for more expressive functionalities, in particular, those that model corruption and exactly resemble the ones that can be found in the cryptographic literature.

Our ideal functionality IdealPKE : $I_{PKE}$ is defined on top of the interface $I_{Enc}$ (which, in a complete system, is implemented by the environment or the simulator), i.e. $I_{Enc} \vdash$ IdealPKE, where $I_{Enc}$ is as follows:

```
class KeyPair {)
    public byte[] publicKey;
    public byte[] privateKey;
}
class Encryption {
    static public KeyPair generateKeyPair();
    static public byte[]
        encrypt(byte[] publKey, byte[] message);
    static public byte[]
        decrypt(byte[] privKey, byte[] message);
}
```

In a nutshell, IdealPKE works as follows: On initialization, via `Decryptor()`, a public/private key pair is created by calling `generateKeyPair()` in $I_{Enc}$. IdealPKE also maintains a list of message/ciphertext pairs; this list is shared with all associated encryptors (objects returned by `getEncryptor`). When method `encrypt` in $I_{PKE}$ is called for a message $m$, the ideal functionality calls `encrypt` in $I_{Enc}$ to encrypt a sequence of zeros of the same length, obtaining the ciphertext $m'$, and stores $(m,m')$ in the list. The method `decrypt` in $I_{PKE}$ called for $m'$ looks for a pair $(m,m')$ in the list and, if it finds it, returns $m$; otherwise, it decrypts $m'$ (calling `decrypt` in $I_{Enc}$) obtaining $m''$ and returns this message. The idea behind this functionality is that the ciphertext returned by encryption is not related in any way (except for the length of the message) to the plaintext.

As already mentioned, RealPKE is defined in a straightforward way using any IND-CCA2-secure encryption scheme. The following theorem says that RealPKE in fact realizes IdealPKE.

**Theorem 6.** *If* RealPKE *uses an IND-CCA2-secure encryption scheme, then* RealPKE $\leq^{(I_{PKE},\emptyset,\emptyset,I_{Enc})}$ IdealPKE.

## IX. Case Study

In our case study and as a proof of concept of our framework, 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 use the *Joana* tool (see below), to verify strong secrecy of the messages sent over the network, i.e., non-interference is shown using Joana for the system when it runs with IdealPKE and by our framework we then obtain computational indistinguishability guarantees when IdealPKE is replaced by RealPKE.

We emphasize that, while the code of client and server are quite small, the actual code that needs to be analyzed is larger because it includes the ideal functionality and the code that results from applying the techniques developed in Section VII-B (we note that the verified program is in the family of systems considered in this section); altogether the code in our case study comprises 376 LoC in 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.

The main goal of our case study is to demonstrate that it is, in principle, possible to establish strong cryptographic security guarantees for Java(-like) programs using existing tools for checking standard non-interference properties.

The verification of the program considered in our case study took just a few seconds (see below), and hence, our approach, in conjunction with Joana, should also work for bigger programs and more complex settings.

### A. The Analyzed Program

We now describe the analyzed program in more detail (see [24] for the code).

Besides the code for the 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 $\texttt{Setup}[b]$, where $b \in \{\texttt{false}, \texttt{true}\}$ is the value with which `secret` is initialized in `Setup`. By $S^{real}[b]$, for $b \in \{\texttt{false}, \texttt{true}\}$, we denote the system consisting of the class $\texttt{Setup}[b]$, 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).

We note that for the analysis with Joana we consider a variant of the ideal functionality where the ideal encryption is done always with the zero byte, and hence, the ideal functionality does not reveal the length of the encrypted message. (This is reasonable if only messages of fixed length are supposed to be encrypted.)

### B. The Property to be Proven

The property we want to show is

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

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 $I$-non-interference of the system $S^{ideal}[b]$. Since the system $S^{ideal}[b]$ is in the class of systems considered in Section VII-B, we can use the results from that section which says that we only need to show (classical) non-interference of the system $T_{\vec{u}}[b]$, for all $\vec{u}$, which extends $S^{ideal}[b]$ by $\tilde{E}_u^{I_E}$, where $I_E$ is the interface IdealPKE extended with methods for network input and output.

To show that, for all $\vec{u}$, non-interference holds for $T_{\vec{u}}[b]$, we used the Joana tool, as described in the next section.

The verification carried out establishes (4), under the (reasonable) assumption that Joana is sound with respect to the subset of Java covered in Jinja+ (as explained in the next section, Joana has been proved to be sound with respect to the semantics of Jinja).

### C. Analysis with Joana

Joana [19] is a static analysis tool. It uses a technique called *slicing* for *program dependence graphs* (PDG)—a graph-based representation of the program—to detect illegal information flows. It can handle full Java bytecode including exceptions. A machine-checked proof [33] provides a formal specification of PDGs and shows that slicing can be used to obtain a sound approximation of the information flows inside a program. Additional work [34], [35] verified that a variant of the slicing algorithm used by Joana can help to guarantee classical Goguen/Meseguer non-interference [16] (which is the kind of non-interference we are interested in) for the semantics of the Jinja language. The technique used by Joana does not depend on such details of the semantics as the maximum value of integers. Therefore, the guarantees Joana gives apply to all variants of the semantics that differ on this maximum value.

Joana is a whole-program analysis tool that analyzes an explicit version of a program and thus cannot reason about the security of a family of programs in general. We show, however, that this is possible for the specific families of (closed) programs $\tilde{E}_{\vec{u}} \cdot S$ (parametrized by $\vec{u}$), considered in Section VII-B. In particular, Joana can verify (absence of) information flow for the family $T_{\vec{u}}$ defined in the previous section. We want to emphasize that these results are not specific to the protocol analyzed in this case study; they enable Joana to reason about any system that complies with the restrictions of Section VII-B, and hence, these results are of general interest.

In the verification process we have carried out, we have marked the initialization of the variable `secret` as high input and modifications to the `result` variable of `unstrustedOutput` as low output. Then Joana automatically has built the PDG model of the program, marked the corresponding nodes in the graph with high and low, and checked if no information flow is possible from high input to low output through a data flow analysis on the graph. (In case an illegal flow is detected Joana issues a violation of the security property and returns the set of all possible paths (*slice*) of illegal flow in the program.)

Because of various precision enhancements in Joana—especially detection of impossible `NullPointerExceptions` and object-sensitivity—we were able to analyze the given family of programs (without any false positives that might have stemmed from the involved overapproximations) and thus to guarantee the absence of information flow.

Joana took about 11 seconds on a standard PC to finish the analysis of the program (with a size of 376 LoC). PDG computation took 10 seconds and only 1 second was needed to detect the absence of illegal flow inside the PDG.

## X. Related Work

As already mentioned in the introduction, our work contributes to the area of language-based analysis of software that uses cryptography, such as cryptographic protocols, an area that recently has gained much attention. We discuss some of the more closely related work in this area in what follows.

The work most closely related to our work is the work by Fournet et al. [14]. Fournet et al. also aim at establishing computational indistinguishability properties for systems written in a practical programming language; this work, in fact, seems to be the first to study such strong properties in the area of language-based cryptographic analysis, other work, including the work mentioned below, considers trace properties, such as authentication and weak secrecy. However, there are many differences between the work by Fournet et al. and our work, in terms of the results obtained, the approach taken, and the programming language considered. Fournet et al. consider a fragment of the functional language F#, while we consider a fragment of Java. Their approach, with theorems of the form "if a program type-checks, then it has certain cryptographic properties", is strongly based on type checking (with refinement types), while the point of our framework is to enable different techniques and tools that a priori cannot deal with cryptography to establish cryptographic guarantees. While the results of Fournet et al. concern specific cryptographic primitives, we establish general results for ideal functionalities and their realizations. While simulation-based techniques are used in the *proofs* of the theorems in the work of Fournet et al., in our approach, ideal functionalities are part of the program to be analyzed by the tools; by our framework, the ideal functionalities can then be replaced by their realizations and for the resulting systems we obtain computational indistinguishability.

In most of the work on language-based analysis of crypto-based software the analysis is carried out based on a symbolic (Dolev-Yao), rather than a cryptographic model (see, e.g., [18], [6], [11], [8]). Some works obtain cryptographic guarantees by applying computational soundness results [5], [2], where the usual restrictions of computational soundness results apply [13], or by compiling the source code to a specification language that then can be analyzed by a specialized tool, namely CryptoVerif [9], for cryptographic analysis [7]. In contrast, our approach, just as the one by Fournet et al. discussed above, does not take a detour through symbolic models in combination with computational soundness results. Also, our approach does not rely on specialized tools for cryptographic analysis.

The existing works on the security analysis of crypto-based software has mostly focussed on fragments of F# and C, including the above mentioned works. Some works consider (fragments of) Java [21], [30], but in a symbolic model and without formal guarantees.

Our framework might also be applicable in the context of computational non-interference (see, e.g., [15], [26]) since computational non-interference can be seen as a specific form of computational indistinguishability. It is interesting future work to investigate the connection between these works and our work further.

Our work also contributes to the mostly unexplored field of non-interference for interactive/open systems [29], [12]. Our technique presented in Section VII enables program analysis tools for checking non-interference of closed systems to deal with open/interactive systems in a practical programming language, namely Java. Existing works on non-interference for interactive systems [29], [12] are orthogonal to our work in that on the one hand they consider abstract system models (labeled transition systems with input and output), rather than a practical programming language, and on the other hand they study systems w.r.t. a more general lattice of security labels for input/output channels.

Finally, we mention a line of work that considers methods for generating secure Java programs from abstract models of cryptographic protocols (see, e.g. [4]).

## XI. Conclusion

In this paper, we have presented a general framework for establishing computational indistinguishability properties for Java(-like) programs using program analysis tools that can check (standard) non-interference properties of Java programs but a priori cannot deal with cryptography and cryptographic adversaries, i.e., probabilistic polynomial-time adversaries. The approach we have proposed is new and combines techniques from program analysis and simulation-based security. Our framework is stated and proved for the Java-like language Jinja+, which comprises a rich fragment of Java.

As a proof of concept of our framework, we have used an automatic tool, namely Joana, to check the non-interference of a Java program. By our framework, this analysis implies computational indistinguishability for that program (w.r.t. an active adversary). The analysis performed by Joana was very fast and suggests that more complex systems can be analyzed within our framework using this tool. Our case study has thus demonstrated, for the first time, that general program analysis tools that a priori are not designed to perform

cryptographic analysis of Java programs, can in fact be used for that purpose.

Our work contributes to the field of language-based analysis of crypto-based software, which recently has gained much attention, in that i) a new approach is proposed, ii) our approach works for a (rich fragment of a) practical programming language, namely Java, which has not gotten much attention so far in this area, and iii) computational indistinguishability guarantees are obtained, rather than only guarantees in more abstract symbolic (Dolev-Yao) models and rather than trace properties, such as authentication and weak secrecy, as in most other works, and iv) these guarantees are obtained directly without taking a detour through symbolic models in combination with computational soundness results, as in most other works.

There are many directions for future work. We briefly mention a few. First, we are confident that, besides Joana, also other program analysis tools can be used within our framework to establish cryptographic security properties of Java programs, with possible candidates being the interactive theorem prover KeY [1], a tool based on Maude [3], and Jif [27], [28].

**Acknowledgment.** This work was partially supported by *Deutsche Forschungsgemeinschaft* (DFG) under Grant KU 1434/6-1 within the priority programme 1496 "Reliably Secure Software Systems – RS$^3$".

## REFERENCES

[1] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and System Modeling*, 4:32–54, 2005.

[2] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. Extracting and verifying cryptographic models from C protocol code by symbolic execution. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011)*, pages 331–340. ACM, 2011.

[3] Mauricio Alba-Castro, María Alpuente, and Santiago Escobar. Abstract Certification of Global Non-interference in Rewriting Logic. In Frank S. de Boer, Marcello M. Bonsangue, Stefan Hallerstede, and Michael 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.

[4] Matteo Avalle, Alfredo Pironti, Riccardo Sisto, and Davide Pozza. The JavaSPI framework for security protocol implementation. In *Availability, Reliability and Security (ARES 11)*, page 746–751. IEEE Computer Society, IEEE Computer Society, 2011.

[5] Michael Backes, Matteo Maffei, and Dominique Unruh. Computationally sound verification of source code. In Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov, editors, *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS 2010)*, pages 387–398. ACM, 2010.

[6] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse. Verified Interoperable Implementations of Security Protocols. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW-19 2006)*, pages 139–152. IEEE Computer Society, 2006.

[7] Karthikeyan Bhargavan, Cédric Fournet, Ricardo Corin, and Eugen Zalinescu. Cryptographically verified implementations for TLS. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *Proceedings of the 2008 ACM Conference on Computer and Communications Security (CCS 2008)*, pages 459–468. ACM, 2008.

[8] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. Modular verification of security protocol code by typing. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010)*, pages 445–456. ACM, 2010.

[9] B. Blanchet. A Computationally Sound Mechanized Prover for Security Protocols. In *IEEE Symposium on Security and Privacy (S&P 2006)*, pages 140–154. IEEE Computer Society, 2006.

[10] 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.

[11] S. Chaki and A. Datta. ASPIER: An automated framework for verifying security protocol implementations. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF 2009)*, pages 172–185. IEEE Computer Society, 2009.

[12] David Clark and Sebastian Hunt. Non-Interference for Deterministic Interactive Programs. In Pierpaolo Degano, Joshua D. Guttman, and Fabio Martinelli, editors, *Formal Aspects in Security and Trust, 5th International Workshop, FAST 2008, Revised Selected Papers*, volume 5491 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 2008.

[13] Hubert Comon-Lundh and Véronique Cortier. How to prove security of communication protocols? A discussion on the soundness of formal models w.r.t. computational ones. In Thomas Schwentick and Christoph Dürr, editors, *Proceedings of the 28th International Symposium on Theoretical Aspects of Computer Science (STACS 2011)*, volume 9 of *LIPIcs*, pages 29–44. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[14] Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular code-based cryptographic verification. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011)*, pages 341–350. ACM, 2011.

[15] Cédric Fournet, Jérémy Planul, and Tamara Rezk. Information-flow types for homomorphic encryptions. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011)*, pages 351–360. ACM, 2011.

[16] J. Goguen and J. Meseguer. Interference Control and Unwinding. In *Proc. Symposium on Security and Privacy*, pages 75–86. IEEE, 1984.

[17] Joseph A. Goguen and José Meseguer. Security Policies and Security Models. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[18] J. Goubault-Larrecq and F. Parrennes. Cryptographic protocol analysis on real C code. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005*, volume 5, pages 363–379. Springer, 2005.

[19] Christian Hammer and Gregor 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.

[20] Dennis Hofheinz, Dominique Unruh, and Jörn Müller-Quade. Polynomial Runtime and Composability. Technical Report 2009/023, Cryptology ePrint Archive, 2009. Available at http://eprint.iacr.org/2009/023.

[21] Jan Jürjens. Security Analysis of Crypto-based Java Programs using Automated Theorem Provers. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*, pages 167–176. IEEE Computer Society, 2006.

[22] Gerwin Klein and Tobias Nipkow. A Machine-Checked Model for a Java-Like Language, Virtual Machine, and Compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, 2006.

[23] 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.

[24] Ralf Küsters, Tomasz Truderung, and Jürgen Graf. A Framework for the Cryptographic Verification of Java-like Programs. Cryptology ePrint Archive, Report 2012/153, 2012. http://eprint.iacr.org/2012/153.

[25] Ralf Küsters and Max Tuengerthal. Joint State Theorems for Public-Key Encryption and Digitial Signature Functionalities with Local Computation. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF 2008)*, pages 270–284. IEEE Computer Society, 2008.

[26] Peeter Laud. Semantics and Program Analysis of Computationally Secure Information Flow. In David Sands, editor, *Programming Languages and Systems, 10th European Symposium on Programming, ESOP 2001*, volume 2028 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2001.

[27] A.C. 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.

[28] Andrew C. Myers, Stephen Chong, Nathaniel Nystrom, Lantien Zheng, and Steve Zdancewic. *Jif: Java Information Flow (software release)*, July 2001. http://www.cs.cornell.edu/jif/.

[29] Kevin R. O'Neill, Michael R. Clarkson, and Stephen Chong. Information-Flow Security for Interactive Programs. In *19th IEEE Computer Security Foundations Workshop (CSFW-19 2006)*, pages 190–201. IEEE Computer Society, 2006.

[30] Nicholas O'Shea. Using Elyjah to Analyse Java Implementations of Cryptographic Protocols. In *Joint Workshop on Foundations of Computer Security, Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security (FCS-ARSPA-WITS-2008)*, 2008.

[31] 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.

[32] Andrei Sabelfeld and Andrew 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.

[33] Daniel Wasserrab. *From Formal Semantics to Verified Slicing - A Modular Framework with Applications in Language Based Security*. PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik, October 2010.

[34] Daniel Wasserrab and Denis Lohner. Proving Information Flow Noninterference by Reusing a Machine-Checked Correctness Proof for Slicing. In *6th International Verification Workshop - VERIFY-2010*, July 2010.

[35] Daniel Wasserrab, Denis Lohner, and Gregor Snelting. On PDG-Based Noninterference and its Modular Proof. In *Proceedings of the 4th Workshop on Programming Languages and Analysis for Security*, pages 31–44. ACM, June 2009.