A Framework for the Cryptographic Verification of Java-like Programs

Technical Report

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) noninterference 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 noninterference 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) noninterference properties [16] of Java programs, such as the tools Joana [19], KeY [1], a tool based on Maude [3], and Jif [28], [29]. 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 noninterference properties cannot deal with cryptography directly. In particular, they typically do not 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., [35] 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., [9], [34], [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) and 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. 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) noninterference [35] is equivalent to perfect indistinguishability, we obtain that by proving noninterference using existing program analysis tools, we can establish computational indistinguishability properties.

Many program analysis tools can only deal 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. 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 proof noninterference 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., [9], [24]), 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 noninterference properties of Java programs, we show that our system enjoys the noninterference 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 noninterference is shown in Sections V and VI, with the proof technique for noninterference in open systems discussed in Section VII. The ideal functionality for public-key encryption and its realization can be found in Section 7, with the case study presented in Section IX. In Section X, we discuss related work. We conclude in Section XI. More details are provided in the appendix.

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.

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 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. 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 Appendix B.

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*.

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 Appendix B.

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 (see Appendix B-B for details).

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., [9], [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 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.

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 send over tapes between different machines, objects can be created, data of different types, including references pointing to possibly complex objects, can be passed between different objects, and classes of the another subsystem can be extended by inheritance. 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 $E \cdot S \rightsquigarrow \texttt{true}$ 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`.

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 η, we define a system SP_η that implements the interface I_{SP} by setting the initial value of `securityParameter` to η. 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 will 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, we will let the size of integers grow with the security parameter.

D. Perfect Indistinguishability

We now formulate (termination-insensitive) 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 η 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 $E \cdot S_2(\eta) \rightsquigarrow_s \texttt{true}$.

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 η as input and outputs a natural number ≥ 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, [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 $intsize(\eta)$) exceeds $p(\eta)$ is a negligible function in η.[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 $intsize(\eta)$) 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 $intsize(\eta)$), 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*).

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*,

[1]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.

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 η.

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 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 (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 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. 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. To prove these results, we need the following lemma, with the proof provided in Appendix C-A.

Lemma 1. *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 $E \cdot S_1 \equiv_{\mathsf{comp}} E \cdot S_2$.*

Now we can prove reflexivity and transitivity of strong simulatability.

Lemma 2 (**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.*

Proof: We define $S = \emptyset$ and immediately obtain that $R \approx^{I_{out}}_{\mathsf{comp}} R = S \cdot R$. ■

Lemma 3 (**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 $R_2 \leq^{(I_{out},I_{in},I_E,I_S^0 \cup I_S^1)} R_0$.

Proof: Under the assumption of the lemma, we know that there exist S_0 and S_1 such that $I_E \cup I_S^1 \vdash S_0 : I_S^0$, $I_E \vdash S_1 : I_S^1$, $S_0 \cdot R_0$ and $S_1 \cdot R_1$ are environmentally I_{out}-bounded, and $R_1 \approx_{\mathsf{comp}}^{I_{out}} S_0 \cdot R_0$ and $R_2 \approx_{\mathsf{comp}}^{I_{out}} S_1 \cdot R_1$. We define $S = S_0 \cdot S_1$. Obviously, we have that $I_E \vdash S : I_S^0 \cup I_S^1$. Now let E be a bounded I_{out}-environment for R_2. Then, we obtain:

$$E \cdot R_2 \equiv E \cdot S_1 \cdot R_1 \equiv E \cdot S_1 \cdot S_0 \cdot R_0 = E \cdot S \cdot R_0 .$$

The first equivalence holds because of our assumptions. For the second equivalence, first note that $E \cdot S_1 \cdot R_1$ is almost bounded and $R_1 \approx_{\mathsf{comp}}^{I_{out}} S_0 \cdot R_0$. By Lemma 1, we now obtain that $E \cdot S_1 \cdot S_0 \cdot R_0$ is almost bounded and that the second equivalence holds. ■

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, $R_0 \cdot R_1 \leq^{(I_0 \cup I_1,\emptyset,I_E,I_S^0 \cup I_S^1)} F_0 \cdot F_1$.*

Proof: Under the assumptions of the theorem, there exist S_0 and S_1 such that $I_E \vdash S_i : I_S^i$, $S_i \cdot F_i$ is environmentally I_i-bounded and $R_i \approx_{\mathsf{comp}}^{I_i} S_i \cdot F_i$ for $i \in \{0,1\}$.

We define $S = S_0 \cdot S_1$, $R = R_0 \cdot R_1$, $F = F_0 \cdot F_1$, $I = I_0 \cup I_1$, and $I_S = I_S^0 \cup I_S^1$. In order to show that $R \leq^{(I,\emptyset,I_E,I_S)} F$ it suffices to prove that a) $S \cdot F$ is environmentally I-bounded and b) $R \approx_{\mathsf{comp}}^{I} S \cdot F$; the remaining conditions are obvious.

Let E be a bounded I-environment for R. Similarly to the proof of Lemma 3, by Lemma 1 we obtain the following equivalences:

$$\begin{aligned} E \cdot R &= E \cdot R_0 \cdot R_1 \\ &\equiv E \cdot R_0 \cdot (S_1 \cdot F_1) \\ &\equiv E \cdot (S_0 \cdot F_0) \cdot (S_1 \cdot F_1) = E \cdot S \cdot F, \end{aligned}$$

This implies b). By Lemma 1 we, in particular, get that all the above systems are almost bounded. Since we quantified over all bounded I-environments for R (and hence, $S \cdot F$) it follows that $S \cdot F$ is environmentally I-bounded, thus a). ■

Note that with $R_0 \cdot R_1 \leq^{(I_0 \cup I_1,\emptyset,I_E,I_S^0 \cup I_S^1)} F_0 \cdot F_1$ we in particular have the realization for all subinterfaces of $I_0 \cup I_1$.

For simplicity, the theorem 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.

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, $F \cdot P_1 \approx_{\mathsf{comp}}^{J} F \cdot P_2$ implies $R \cdot P_1 \approx_{\mathsf{comp}}^{J} R \cdot P_2$.*

Proof: Assume that $F \cdot P_1 \approx_{\mathsf{comp}}^{J} F \cdot P_2$. 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

$$R \approx_{\mathsf{comp}}^{I} S \cdot F \tag{1}$$

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 1 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 $F \cdot P_1 \approx_{\mathsf{comp}}^{J} F \cdot P_2$, by Lemma 1 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, $i \in \{1,2\}$, only uses I as a (trusted) service and F/R uses I_P. It is straightforward to also allow for other external services.

We now show that perfect indistinguishability implies computational indistinguishability (see Appendix C-B) for details.

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 $intsize(\eta)$. 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 Noninterference

In this section we prove that perfect indistinguishability and noninterference are equivalent for an appropriate class of systems. Hence, in combination with Corollary 1 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. As already mentioned in the introduction, many tools can analyze noninterference for Java programs.

The (standard) noninterference notion for confidentiality [16] requires the absence of information flow from high to low variables within a program. In this paper, we define noninterference 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 initiated with values $\vec{a}$ and the low variables are initiated 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, noninterference for a (deterministic) program is defined as follows.

Definition 17 (Noninterference for Jinja+ programs). Let $P[\vec{x}]$ be a program with high and low variables. Then, $P[\vec{x}]$ *has the noninterference 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.

We note that the noninterference 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 noninterference property can be as powerful as perfect indistinguishability.

The above notion of noninterference deals with complete programs (closed systems). We now lift this definition to open systems.

Definition 18 (Noninterference 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 η noninterference 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-noninterference 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-noninterference 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 noninterference for systems that only use ideal functionalities.

VII. A Proof Technique for Noninterference in Open Systems

There are many tools that can deal with classical noninterference (noninterference of a complete program). In our case study, we demonstrate that at least one of these tools can also deal with noninterference in certain open systems. In the remainder of this section we develop some techniques that help with this process and that should also enable other tools to deal with relevant open systems. We start with a simple case when the communication between P and E is restricted to primitive types only. Then we generalize our result to the case where P and E 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

```
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.

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.

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.

First, $\tilde{E}_{\vec{u}}^{I_E}$ contains static methods: `untrustedOutput` and `untrustedInput`, as specified in Figure 2. 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` will get, as we will see below, all the data the environment gets from S. If the two variants of S (with different high values) behave differently, then there must be some point where the environment gets different data from the two systems in the corresponding runs. By choosing an appropriate $\vec{u}$ this can be detected by `untrustedOutput`, which will assign different values to `result`.

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

```
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 different identifier to every method).

forwards all its input data to `untrustedOutput` and let `untrustedInput` determine the returned value.

This completes the definition of $\tilde{E}_{\vec{u}}$. The next theorem (see Appendix D-A for the proof) states that, to prove I-noninterference, it is enough to only consider environments $\tilde{E}_{\vec{u}}$. In this way we only need to study almost closed systems, namely systems that differ in only one expression (line 9).

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-noninterference, for $I = \emptyset$, holds for S if and only if for all sequences $\vec{u}$ as above noninterference 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 arrays, simple object, and throwing exceptions. Some restrictions, however, have to be imposed on I_E and the program S. These restrictions guarantee that, although references are exchanged between E and S, the communication resembles exchange of pure data.

More precisely, our result works for the following class of systems S. Let I_E be the minimal interface such that $I_E \vdash S$. We assume that:

1. Methods of classes in I_E are static and their arguments are of primitive types or of type `byte[]`. (Let us notice that this only restricts the way S uses the environment; an environment that implements the interface I_E can have arbitrary fields and methods. Note also that methods of I_E can throw exceptions.)
2. Fields of classes in I_E are non-static and of primitive types or of type `byte[]`. (Again, a system implementing I_E is not restricted to such fields).
3. Whenever an array (i.e. the reference to an array) is passed to the environment, this reference is not used by S afterwards. (This property can be easily guaranteed by a syntactical restriction to pass only fresh copies of arrays to the environment).
4. If S calls a (static) method of I_E which returns a reference r, the way S can use r is subject to the following

```
static public byte[] foo(int x, byte t[]) throws C {
    // consume the input:
    untrustedOutput(FOO_ID);
    untrustedOutput(x);
    untrustedOutput(t.lenght);
    for( int i=0; i<t.length; ++i )
        untrustedOutput(t[i]);
    // decide whether to throw some exception:
    if(untrustedInput()==0) throw new C_1();
    ...
    if(untrustedInput()==0) throw new C_k();
    // determine the array to return:
    int len = untrustedInput();
    if (len<0) return null;
    byte[] result = new byte[len];
    for( int i=0; i<len; ++i)
        result[i] = (byte) untrustedInput();
    return result;
}
```

Figure 4. $\tilde{E}_{\vec{u}}$: the implementation of a method with the signature `static public byte[] foo(int x, byte[] t) throws C`, where $C_1,\ldots,C_k$ are all subclasses of `C`, including `C` itself, which are either standard exceptions or empty classes in I_E.

restrictions: (1) If r is a reference of type `byte[]`, then S is only allowed to immediately clone r; r is not used afterwards. (2) If r is a reference to an object, then S is only allowed to immediately clone the fields of r (of primitive types and of type `byte[]`); r is not used afterwards. S cannot use r in any way not specified explicitly above.

5. For every try-catch statement in S of the form

   `try { ... } catch (C  r) { B }`

   if C or a subclass of C is listed in the `throws` clause of some method in I_E (and thus this statement may potentially catch an exception thrown by E), then r is subject to the same restrictions as in item 4. case (2) above.

For such programs we can, as in the previous section, construct a fixed $\tilde{E}_{\vec{u}}$ such that it is enough to consider only these system (for all $\vec{u}$). This system consists of the static methods `untrustedInput` and `untrustedOutput` as defined above and, for every class C of I_E, the declaration of C with the implementation of (static) methods as illustrated by the example given in Figure 4. This example illustrates the case when an array is returned. When an object of class D is to be returned, then a fresh object of this class is created and the values of its fields that are specified in I_E are filled using `untrustedInput`, as for the array in the example.

The following theorem is a generalisation of Theorem 5 to the richer family of progarms considered in this section (see Appendix D-B for the proof).

Theorem 6. *Let S be as above. I-noninterference holds for S if and only if for all sequences $\vec{u}$ as above noninterference holds for $\tilde{E}_u \cdot S$.*

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, based on which our example program will be analyzed (see Section IX); details of IdealPKE are given in Appendix A-A. We also 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., [9], [24]). 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 7) 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, which says that RealPKE in fact realizes IdealPKE, is proved in Appendix E.

Theorem 7. *If* RealPKE *uses an IND-CCA2-secure encryption scheme, then* RealPKE $\leq^{(I_{PKE},\emptyset,\emptyset,I_{Enc})}$ IdealPKE.

IX. Case Studies

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., noninterference 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.

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 developed in Section VII-B (we note the verified program is in the family of systems considered in this section); altogether the code, which 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 verification of the program 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 Appendix A-B for the code).

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 $\texttt{Setup}[b]$, 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

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

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 VII-B, we can use the results from this section which say that we only need to show (classical) noninterference of the system $T_{\vec{u}}[b]$, for all $\vec{u}$, which extends $S^{ideal}[b]$ by $\tilde{E}^{I_E}_{u}$, where I_E is the interface IdealPKE extended with methods for network input and output.

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

The carried out verification 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 [36] 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 [37], [38] verified that a variant of the slicing algorithm used by Joana can help to guarantee classical Goguen/Meseguer noninterference [15] (which is the kind of noninterference 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 analyses 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 (see Appendix F for details). 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 the results of Appendix F are not specific to the protocol analyzed in this case study; they enable Joana to reason about any systems that comply with the restrictions of Section VII-B, and hence they 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.

The complete source code that was analyzed can be download from [26].

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. [13]. 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 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., [17], [5], [10], [7]). Some works obtain cryptographic guarantees by applying computational soundness results [4], [2], where the usual restrictions of computational soundness results apply [12], or by compiling the source code to a specification language that then can be analyzed by a specialized tool, namely CryptoVerif [8], for cryptographic analysis [6]. 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], [32], but in a symbolic model and without formal guarantees.

Our framework might also be applicable in the context of computational noninterference (see, e.g., [14], [27]) since computational noninterference 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 noninterference for interactive/open systems [31], [11]. Our technique presented in Section VII enables program analysis tools for checking noninterference of closed systems to deal with open/interactive systems in a practical programming language, namely Java. Existing works on noninterference for interactive systems [31], [11] 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.

XI. Conclusion

We have presented a general framework for establishing computational indistinguishability properties for Java(-like) programs using program analysis tools that can check (standard) noninterference properties of Java programs but a priori cannot deal with cryptography and cryptographic adversaries, i.e., probabilistic polynomial-time adversaries. The approach we 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, the usefulness of our framework was demonstrated in a case study, where we used an automatic tool, namely Joana, to check the noninterference of a Java program. By our framework, this analysis implied 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 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 [28], [29].

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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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/.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] J. Goguen and J. Meseguer. Interference control and unwinding. In *Proc. Symposium on Security and Privacy*, pages 75–86. IEEE, 1984.

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

[17] 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.

[18] Jürgen Graf. Speeding up context-, object- and field-sensitive sdg generation. In *9th IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 105–114, September 2010.

[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. 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] R. Küsters and M. 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.

[25] R. Küsters and M. Tuengerthal. Universally Composable Symmetric Encryption. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF 2009)*, pages 293–307. IEEE Computer Society, 2009.

[26] Ralf Küsters, Tomasz Truderung, and Jürgen Graf. Source Code for Simple Protocol Case Study with Joana, 2012. Available at http://infsec.uni-trier.de/publications/software/simplProtJoana.zip.

[27] 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 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2028 of *Lecture Notes in Computer Science*, pages 77–91. Springer, 2001.

[28] 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.

[29] 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/.

[30] Tobias Nipkow and David von Oheimb. Java$_{light}$ is Type-Safe — Definitely. In *POPL*, pages 161–170, 1998.

[31] 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.

[32] 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.

[33] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *SDE 1: Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, pages 177–184, New York, NY, USA, 1984. ACM.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

Appendix A. Programs

A. Public Key Encryption

Real Functionality. By RealPKE we denote the system consisting of the following classes and the system CCA2Enc.

```java
class Encryptor {
  private byte[] publKey = null;

  Encryptor(byte[] pubk)
    { publKey = pubk; }
  public byte[] getPublicKey()
    { return publKey; }
  public byte[] encrypt(byte[] message)
    { return Encryption.encrypt(publKey, message); }
}
class Decryptor {
  private byte[] privKey;
  private byte[] publKey = null;

  public Decryptor() {
      KeyPair kp = Encryption.generateKeyPair();
      privKey = kp.privateKey;
      publKey = kp.publicKey;
  }
  public Encryptor getEncryptor()
      { return new Encryptor(publKey); }
  public byte[] decrypt(byte[] message)
      { return Encryption.decrypt(privKey, message); }
}
```

Ideal Functionality. By IdealPKE we denote the following system.

```java
public final class Decryptor {
  private byte[] privKey;
  private byte[] publKey;
  private MessagePairList log;

  public Decryptor() {
    KeyPair keypair = CryptoLib.generateKeyPair();
    publKey = copyOf(keypair.publicKey);
    privKey = copyOf(keypair.privateKey);
  }
  public Encryptor getPublicInterface() {
    return new Encryptor(log,publKey);
  }
  public byte[] decrypt(byte[] message) {
    byte[] messageCopy = copyOf(message);
    if (!log.contains(messageCopy)) {
      return copyOf(
        Encryption.decrypt(copyOf(privKey),
        messageCopy) );
    } else {
        return copyOf( log.lookup(messageCopy) );
    }
  }
}

```

```
public final class Encryptor {
  private MessagePairList log;
  private byte[] publKey;

  Encryptor(MessagePairList mpl, byte[] publicKey) {
    log = mpl;
    publKey = publicKey;
  }

  public byte[] getPublicKey() {
    return copyOf(publKey);
  }

  public byte[] encrypt(byte[] message) {
    byte[] messageCopy = copyOf(message);
    byte[] randomCipher = copyOf(
        CryptoLib.encrypt(getZeroMessage(message.length),
                          copyOf(publKey)));
    if( randomCipher == null ) return null;
    log.add(messageCopy, randomCipher);
    return copyOf(randomCipher);
  }
}
```

We omit here the declarations of (a) method `getZeroMessage` that for a integer n returns a message of length n consisting of zeros, (b) method `copyOf` that returns a copy of the given message, (c) class `MessagePairList` that stores pairs of messages an offers natural methods: `add` (to add a message pair), `contains` (to check, for a message m, whether there is a pair (m',m) in the list), and `lookup` (which, for a message m, returns m' as above, if it exists). The declaration of these methods can be found in [26].

B. *The Case Study*

We provide here complete code of the essential components (classes) of our case study: the client, the server, and the setup.

```
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);
  }
}
```

```
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);
    }
  }
}
```

This program uses class Network (which we do not provide, as it is part of the environment) with the following interface:

```
class NetworkError extends Exception {

class Network {
  public static void networkOut(byte[] outEnc)
                      throws NetworkError;
  public static byte[] networkIn()
                      throws NetworkError;
}
```

Appendix B.
Jinja+

A. *Jinja+ Extensions*

As a basis of our formal results we take language Jinja+ 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 (that declare which exceptions can be thrown by a method).

For the last three extensions—access modifiers, final classes, and `throws` clauses—we assume that they are provided by a compiler that, first, ensures that the policies expressed by access modifiers, the final modifier, and `throws` clauses are respected and then produces pure Jinja+ code (without access modifiers, the final modifier, and `throws` clauses). In the similar manner we can deal with constructors: a program using constructors can be easily translated to one without constructors (where creation and initialisation of an object is split into two separate steps).

The remaining extensions are described below:

Primitive types. The Jinja language, as specified in [22], offers only boolean and integer primitive types. For our purpose, we find it useful to also include type `byte` with natural conversions from and to `int`. Also, the set of operators on primitive types is extended to include the standard Java operators (such as multiplication). This extensions can be done in very straightforward way and, thus, we skip its detailed description.

Arrays. We will consider only one-dimensional arrays (an extension to multi-dimensional arrays is then quite straightforward; moreover multi-dimensional arrays can be simulated by nested arrays). To extend the Jinja language with one-dimensional arrays, we adopt the approach of [30].

First, we extend the set of types to include array types of the form τ`[]`, where τ is a type. Next, we extend the set of expressions by: (a) creation of new array: `new` τ`[`e`]`, where e is an expression (that is supposed to evaluate to an integer denoting the size of the array) and τ is a type, (b) array access: e_1`[`e_2`]`, (c) array length access: e`.length`, and (d) array assignment: e_1`[`e_2`] :=` e_3.

For this extension, following [30], we redefine a *heap* to be a map from references to *objects*, where an *object* is either an *object instance*, as defined above, or an *array*. An *array* is a triple consisting of its component type, its length l, and a table mapping $\{0,\ldots,l-1\}$ to values.

Extending (small step) semantic rules to deal with arrays is quite straightforward (see the appendix).

The `abort` primitive. Expression `abort`, when evaluated, causes the program to stop. (Technically this expression cannot be reduced and causes the program execution to get stuck.)

Static methods and fields. Fields and methods can be declared as static. However, as can be seen below, to keep the semantics of the language simple, we impose some restrictions on initializers of static fields.

A static method does not require an object to be invoked. The syntax of static method call is `C.f(`*args*`)`, where `C` is the name of a class that provides `f`.

Extending Jinja with with static methods is straightforward. The rule for static method invocation is very similar to the one for non-static method invocation: the difference is that the variable **`this`** is not added to the context (block) within which the method body is executed (a static method cannot reference non-static fields and methods).

We assume that static fields can be initialized only with literals (constants) of appropriate types. If there is no explicit initializer, then a static variable is initialized with the default value of its type. For example, while **`static int`** `x = 7` and **`static int`**`[] t` are valid declarations, the declaration **`static`** `A a =` **`new`** `A()` and **`static int`** `y = A.foo()` are not.

Dealing with more general static initializers is not difficult in principle, but it would require a precise—and quite complicated—model of the initialisation process, the complication we want to avoid.

Extending Jinja with static filed requires only a very little overhead: for a static field `f` declared in class `C` we introduce a global variable `C.f` (note that names of this form do not interfere with names of local variables and method parameters). These global variables are initialized before actual program (expression) is executed, as described in the definition of a run below.

Exceptions. A method declaration can contain a `throws` clause in which classes of exceptions that can be propagated by the method are listed. Such a clause can be omitted, in which case the above mentioned list is considered empty. When the meaning of `throws` clauses is considered, standard subtyping rules are applied (if class A is listed in such a clause, then the method can propagate exceptions of class A or any subclass of A).

As mentioned, we assume that the compiler (or a static verifier) statically checks whether the program complies with `throws` clauses.

Unlike in Java, however, we can assume without loss of generality that all exceptions must be declared in a `throws` clause if they are propagated by a method (in the Java terminology, we can say that all exceptions are checked). This will give us more control on the information which is passed between program components.

We consider the following hierarchy of standard (system) exceptions. In the root of this hierarchy we place (empty) class `Exception`. We require that only object of this class (and its subclasses) can be used as exceptions. Class `SystemException`, also empty, is a subclas of class `Exception`, and is a base class for the following system exceptions (exceptions which are not thrown explicitly, but may occur in result of some standard operations on expressions):

`ArrayStoreException` — trown to indicate an attempt to store an object of the wrong type into an array,

`IndexOutOfBoundsException` — thrown to indicate that an array has been indexed with an index being out of range,

`NegativeArraySizeException` — thrown to indicate an attempt to create an array with negative size,

`NullPointerException` — thrown if the `null` reference is used when an object is required,

`ClassCastException` — thrown to indicate an illegal cast.

We will assume that the above classes are predefined, and can be used in any program.

For completeness of the presentation, in this section we summarize all the rules of Jinja+. We start with rules of

Jinja, following [22] (see this paper for the details on the used symbols). In particular, the syntactical convention used in these rules is that an application of a function f to an argument a is denoted by $f\ a$.

The rules assume a function *binop* that provides semantics for operations on atomic types. The exact definition of this function depends on the maximal size of integers that we consider (recall that we consider different variants of semantics for different size of integers given by $intsize(\eta)$ where η is the security parameter).

B. Rules of Jinja

There are two point where our presentation rules diverge from the ones of [22]. First, as we assume unbounded memory, we do not have rules which throw `OutOfMemoryError` (and we assume that (*new-Addr h*) is never *None*). Second, we added labels to rules. These labels allow us to count the number of steps performed within (by) a given class or subsystem. A label D in a step

$$\langle e, s\rangle \xrightarrow{D} \langle e', s'\rangle$$

means, informally, that the step was executed by the code of class D. More precisely, the expression that was selected to be reduced by an elementary rule comes from a method of D. We use the label $-$ if the origin of the reduced expression is not known (because, at that point, the context of this expression is not known; typically this empty label is overwritten by a subexpression reduction rule for blocks, that is rules (12)–(14)).

To define labeling of transitions, labels are also added to blocks that are obtained from the method call rule (a block is labeled by the name of the class from which the body of the method comes). Then, the labels of transitions are, roughly speaking, inherited from the innermost block within which the reduction takes place.

Now, for the run of a program P with a subsystem S, we say that a step $\langle s_1, e_1\rangle \xrightarrow{D} \langle s_2, e_2\rangle$ is performed by S and write $\langle s_1, e_1\rangle \xrightarrow{S} \langle s_2, e_2\rangle$, if D is the name of a class defined in S.

Subexpression reduction rules (Figure 5) describe the order in which subexpressions are evaluated. The relation $[\rightarrow]$ it the extension of $\rightarrow$ to expression list ($\cdot$ is the list constructor).

Expression reduction rules (Figure 6) are applied when the subexpressions are sufficiently reduced. In the rule for method invocation, the required nested block structure is built with the help of the auciliary function *blocks*:

$$\begin{aligned} blocks_C([],[],[],e) &= e \\ blocks_C(V\cdot Vs, T\cdot Ts, v\cdot vs, e) &= \\ &= \{V : T; V := v;\ blocks(Vs, Ts, vs, e)\}_C \end{aligned}$$

(where $\cdot$ is the list constructor and $[]$ denotes the empty list).

Exceptional reduction and *exception propagation* rules (Figure 7 and 8) describe how exception are thrown and propagated.

Note that we do not have a rule reducing `abort`. That means that, if this expression is to be reduced, the execution gets stuck.

C. Rules for Jinja+

In this section we presents additional rules of Jinja+. Theres rules concern static method invocation and arrays. The rules are given in Fiture 10 and 9.

APPENDIX C. PROOFS FOR SIMULATABILITY

A. Proof of Lemma 1

Let I, I_E, S_1, S_2, and E be given as stated in the lemma. We need to show that $E\cdot S_2$ is almost bounded and that $E\cdot S_1 \equiv_{\mathsf{comp}} E\cdot S_2$.

Since $E\cdot S_1$ is almost bounded, there exists a polynomial p such that the probability that the length of the run of this system with security parameter η (and integer size $intsize(\eta)$) exceeds $p(\eta)$ is negligible. Now let us denote by $[E]$ the system that is defined just as E, but which in addition has a private static counter (defined in some new class in E) and where the code of E is modified such that whenever a step in the code of E is performed (according to the small step Jinja+ semantics), then the counter is increased. Once the bound $p(\eta)$ is reached, $[E]$ performs `abort()`.

By construction of $[E]$ it is easy to see that $[E]$ is a bounded environment because $[E]$ does not simulate more than $p(\eta)$ steps of E, where each step of E can be simulated in a number of steps bounded by a constant. Also, $[E]$ behaves exactly like E up to the point where the bound $p(\eta)$ is reached. From this, as further explained below, we obtain:

$$E\cdot S_1 \equiv_{\mathsf{comp}} [E]\cdot S_1 \equiv_{\mathsf{comp}} [E]\cdot S_2 \equiv_{\mathsf{comp}} E\cdot S_2.$$

The first equivalence holds because E reaches the bound $p(\eta)$ when running with S_1 only with negligible probability. Hence, the assignment of E and $[E]$ to `result` is the same with overwhelming probability.

The second equivalence is true because $S_1 \approx^{I}_{\mathsf{comp}} S_2$ and $[E]$ is a *bounded* I-environment for S_1 and S_2.

The third equivalence holds because in the system $E\cdot S_2$ the bound $p(\eta)$ is reached also only with negligible probability: Otherwise, we could easily turn $[E]$ into a bounded environment E' that distinguishes S_1 and S_2, namely, E' works just as $[E]$ but outputs `true`, i.e., assigns `true` to the variable `result` iff the bound $p(\eta)$ is reached. So, if, when E interacts with S_2, the bound were reached with non-negligible probability, E' could distinguish between S_1 and S_2. It follows that $E\cdot S_2$ is almost bounded and that the last equivalence holds.

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle \texttt{Cast}\ C\ e,s\rangle \xrightarrow{\ell} \langle \texttt{Cast}\ C\ e',s'\rangle} \tag{5}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle V := e,s\rangle \xrightarrow{\ell} \langle V := e',s'\rangle} \tag{6}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle e.F\{D\},s\rangle \xrightarrow{\ell} \langle e'.F\{D\},s'\rangle} \tag{7}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle e.F\{D\} := e_2,s\rangle \xrightarrow{\ell} \langle e'.F\{D\} := e_2,s'\rangle} \tag{8}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle \texttt{Val}\ v.F\{D\} := e,s\rangle \xrightarrow{\ell} \langle \texttt{Val}\ v.F\{D\} := e',s'\rangle} \tag{9}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle e \ll bop \gg e_2,s\rangle \xrightarrow{\ell} \langle e' \ll bop \gg e_2,s'\rangle} \tag{10}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle \texttt{Val}\ v_1 \ll bop \gg e,s\rangle \xrightarrow{\ell} \langle \texttt{Val}\ v_1 \ll bop \gg e',s'\rangle} \tag{11}$$

$$\frac{P \vdash \langle e,(h,l(V := None))\rangle \xrightarrow{\ell} \langle e',(h',l')\rangle \quad l'\ V = None \quad \neg\ assigned\ V\ e}{P \vdash \langle \{V : T; e\}_D,(h,l)\rangle \xrightarrow{f(l,D)} \langle \{V : T; e'\}_D,(h',l'(V := l\ V))\rangle} \tag{12}$$

$$\frac{P \vdash \langle e,(h,l(V := None))\rangle \xrightarrow{\ell} \langle e',(h',l')\rangle \quad l'\ V = v \quad \neg\ assigned\ V\ e}{P \vdash \langle \{V : T; e\}_D,(h,l)\rangle \xrightarrow{f(l,D)} \langle \{V : T;\ V := \texttt{Val}\ v; e'\}_D,(h',l'(V := l\ V))\rangle} \tag{13}$$

$$\frac{P \vdash \langle e,(h,l(V \mapsto v))\rangle \xrightarrow{\ell} \langle e',(h',l')\rangle \quad l'\ V = v'}{P \vdash \langle \{V : T; V := \texttt{Val}\ v; e\}_D,(h,l)\rangle \xrightarrow{f(l,D)} \langle \{V : T;\ V := \texttt{Val}\ v'; e'\}_D,(h',l'(V := l\ V))\rangle} \tag{14}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle e.M(es),s\rangle \xrightarrow{\ell} \langle e'.M(es),s'\rangle} \tag{15}$$

$$\frac{P \vdash \langle es,s\rangle\ [\xrightarrow{\ell}]\ \langle es',s'\rangle}{P \vdash \langle \texttt{Val}\ v.M(es),s\rangle \xrightarrow{\ell} \langle \texttt{Val}\ v.M(es'),s'\rangle} \tag{16}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle e; e_2,s\rangle \xrightarrow{\ell} \langle e'; e_2,s'\rangle} \tag{17}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle \texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2,s\rangle \to \langle \texttt{if}\ (e')\ e_1\ \texttt{else}\ e_2,s'\rangle} \tag{18}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle e \cdot es,s\rangle\ [\xrightarrow{\ell}]\ \langle e' \cdot es,s'\rangle} \qquad \frac{P \vdash \langle es,s\rangle\ [\xrightarrow{\ell}]\ \langle es',s'\rangle}{P \vdash \langle \texttt{Val}\ v \cdot es,s\rangle\ [\xrightarrow{\ell}]\ \langle \texttt{Val}\ v \cdot es',s'\rangle} \tag{19}$$

Figure 5. Subexpression reduction rules. We define $f(l,D) = D$, if $l = -$; otherwise $f(l,D) = l$.

$$\frac{\textit{new-Addr } h = a \quad P \vdash C \textit{ has-fields } FDTs}{P \vdash \langle \texttt{new } C, (h,l)\rangle \xrightarrow{-} \langle \textit{addr } a, (h(a \mapsto (C, \textit{ init-fields } FDTs)), l)\rangle} \tag{20}$$

$$\frac{hp\ s\ a = (D, f_s) \quad P \vdash D \preceq^* C}{P \vdash \langle \texttt{Cast } C\ (\textit{addr } a), s\rangle \xrightarrow{-} \langle \textit{addr } a, s\rangle} \tag{21}$$

$$P \vdash \langle \texttt{Cast } C\ \textit{null}, s\rangle \xrightarrow{-} \langle \textit{null}, s\rangle \tag{22}$$

$$\frac{lcl\ s\ V = v}{P \vdash \langle \texttt{Var } V, s\rangle \xrightarrow{-} \langle \texttt{Val } v, s\rangle} \tag{23}$$

$$P \vdash \langle V := \texttt{Val } v, (h,l)\rangle \xrightarrow{-} \langle \textit{unit}, (h, l(V \mapsto v))\rangle \tag{24}$$

$$\frac{\textit{binop } (\textit{bop}, v_1, v_2) = v}{P \vdash \langle \texttt{Val } v_1 \ll \textit{bop} \gg \texttt{Val } v_2, s\rangle \xrightarrow{-} \langle \texttt{Val } v, s\rangle} \tag{25}$$

$$\frac{hp\ s\ a = (C, fs) \quad fs(F,D) = v}{P \vdash \langle \textit{addr } a.F\{D\}, s\rangle \xrightarrow{-} \langle \texttt{Val } v, s\rangle} \tag{26}$$

$$\frac{h\ a = (C, fs)}{P \vdash \langle \textit{addr } a.F\{D\} := \texttt{Val } v, (h,l)\rangle \xrightarrow{-} \langle \textit{unit}, (h(a \mapsto (C, fs((F,D) \mapsto v))), l)\rangle} \tag{27}$$

$$\frac{hp\ s\ a = (C, fs) \quad P \vdash C \textit{ sees } M:\ Ts \to T = (pns,\ body) \textit{ in } D \quad |vs| = |pns| \quad |Ts| = |pns|}{P \vdash \langle \textit{addr } a.M(\textit{map } \texttt{Val } vs), s\rangle \xrightarrow{-} \langle \textit{blocks}_D(\textit{this} \cdot pns,\ \textit{Class } D \cdot Ts,\ \textit{Addr } a \cdot vs,\ body), s\rangle} \tag{28}$$

$$P \vdash \langle \{V : T;\ V := \texttt{Val } v;\ \texttt{Val } u\}_D, s\rangle \xrightarrow{D} \langle \texttt{Val } u, s\rangle \tag{29}$$

$$P \vdash \langle \{V : T;\ \texttt{Val } u\}, s\rangle \xrightarrow{-} \langle \texttt{Val } u, s\rangle \tag{30}$$

$$P \vdash \langle \texttt{Val } v;\ e_2, s\rangle \xrightarrow{-} \langle e_2, s\rangle \tag{31}$$

$$P \vdash \langle \texttt{if}(\textit{true})\ e_1 \texttt{ else } e_2, s\rangle \xrightarrow{-} \langle e_1, s\rangle \tag{32}$$

$$P \vdash \langle \texttt{if}(\textit{false})\ e_1 \texttt{ else } e_2, s\rangle \xrightarrow{-} \langle e_2, s\rangle \tag{33}$$

$$P \vdash \langle \texttt{while}(b)\ c, s\rangle \xrightarrow{-} \langle \texttt{if}(b)\ (c;\ \texttt{while}(b)\ c) \texttt{ else } \textit{unit}, s\rangle \tag{34}$$

Figure 6. Expression reduction

B. Proof of Theorem 3

Let E be a bounded I-environment for S_1 (and hence, S_2). For a finite bit string r, let E_r denote the deterministic system obtained from E by fixing its randomness by r in the following way: The primitive randomBit() is replaced by a method (along with a new static field) declared within E_r such that the first $|r|$ bits returned by the method are chosen according to r; all the remaining bits returned by this method are 0. It follows with $S_1 \approx^I_{\mathsf{perf}} S_2$ that (*) for all security parameters η, for all r, and for all integer sizes $s \geq 1$ such that $E_r \cdot S_1(\eta)$ and $E_r \cdot S_2(\eta)$ terminate for integer size s it holds that $E_r \cdot S_1(\eta) \rightsquigarrow_s \texttt{true}$ iff $E_r \cdot S_2(\eta) \rightsquigarrow_s \texttt{true}$. This implies

$$E \cdot S_1 \equiv_{\mathsf{comp}} E \cdot S_2$$

because: By assumption, $E \cdot S_1$ and $E \cdot S_2$ are almost bounded. Hence, there exists a polynomial p such that the probability that the length of a run (with integer size $\textit{intsize}(\eta)$) of $E \cdot S_1(\eta)$ or $E \cdot S_2(\eta)$ exceeds $p(\eta)$ is negligible. So in all runs, except for a negligible fraction, at most $p(\eta)$ random bits are needed. Moreover, for almost all bit strings r of

$$\frac{hp\ s\ a\ =\ (D,fs) \quad \neg\ P \vdash D \preceq^* C}{P \vdash \langle \texttt{Cast}\ C(addr\ a), s\rangle \xrightarrow{-} \langle \texttt{THROW ClassCastException},\ s\rangle} \tag{35}$$

$$P \vdash \langle null.F\{D\}, s\rangle \xrightarrow{-} \langle \texttt{THROW NullPointerException},\ s\rangle \tag{36}$$

$$P \vdash \langle null.F\{D\} := \texttt{Val}\ v, s\rangle \xrightarrow{-} \langle \texttt{THROW NullPointerException},\ s\rangle \tag{37}$$

$$P \vdash \langle null.M(map\ \texttt{Val}\ vs), s\rangle \xrightarrow{-} \langle \texttt{THROW NullPointerException},\ s\rangle \tag{38}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle \texttt{throw}\ e, s\rangle \xrightarrow{\ell} \langle \texttt{throw}\ e', s'\rangle} \tag{39}$$

$$P \vdash \langle \texttt{throw}\ null, s\rangle \xrightarrow{-} \langle \texttt{THROW NullPointerException}, s\rangle \tag{40}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle \texttt{try}\ e\ \texttt{catch}\ (C\ V)\ e_2, s\rangle \xrightarrow{\ell} \langle \texttt{try}\ e'\ \texttt{catch}\ (C\ V)\ e_2, s'\rangle} \tag{41}$$

$$P \vdash \langle \texttt{try Val}\ v\ \texttt{catch}\ (C\ V)\ e_2, s\rangle \xrightarrow{-} \langle \texttt{Val}\ v, s\rangle \tag{42}$$

$$\frac{hp\ s\ a\ =\ (D,fs) \quad P \vdash\ D\ \preceq^*\ C}{P \vdash \langle \texttt{try THROW}\ a\ \texttt{catch}\ (C\ V)\ e_2, s\rangle \xrightarrow{-} \langle \{V : Class\ C;\ V := addr\ a;\ e_2\}, s\rangle} \tag{43}$$

$$\frac{hp\ s\ a\ =\ (D,fs) \quad \neg\ P \vdash\ D\ \preceq^*\ C}{P \vdash \langle \texttt{try THROW}\ a\ \texttt{catch}\ (C\ V)\ e_2, s\rangle \xrightarrow{-} \langle \texttt{Throw}\ a, s\rangle} \tag{44}$$

Figure 7. Exceptional expression reduction

length at most $p(\eta)$ and integers of size $intsize(\eta)$, we have that the runs of $E_r \cdot S_1(\eta)$ and $E_r \cdot S_2(\eta)$ terminate for integer size $intsize(\eta)$. Now by (*) we know that for such r the output of the runs of $E_r \cdot S_1(\eta)$ and $E_r \cdot S_2(\eta)$ with integer size $intsize(\eta)$ is the same.

Hence $S_1 \approx^I_{\mathsf{comp}} S_2$.

Appendix D. A Proof Technique for Noninterference in Open Systems

A. Proof of Theorem 5

Let I_E and S be like in the theorem. Let $\emptyset \vdash E : I_E$ be an environment for S. We start with some definition that will be useful in the proof.

Below, we consider systems $E \cdot S$ such that the run of $E \cdot S$ is finite. We assume that E and S do not use the `abort` primitive (this assumptions simplifies some notation, but is not crucial; the proof without this assumption is similar).

Let ρ be the run of the system $E \cdot S$. Let (e,s) and (e',s') be configurations in this run. We write $(e,s) \sim (e',s')$ if these configurations are equal up to references (addresses) remapping, i.e. if there exist a bijection f from references to references that $(f(e), f(s)) = (e',s')$, where $f(e)$ and $f(s)$ applies f to every reference in e and, respectively, in s. In the analogous way we define relation $s \sim s'$ on states.

Let $s = (h,l)$ be a state. By $s|_E = (h|_E, l|_E)$ we denote the part of the state that is accessible from E through the static variables that E uses. Formally, we leave in the domain of $l|_E$ only those static variables of l that E can access; we leave in the domain of $h|_E$ only those references that can be reached from those static variables, where a reference can be reached from $l|_E$ if (i) it is stored in one of the variables of $l|_E$ or (ii) it is stored in an object that can be reached from $l|_E$.

In the analogous way we define $s|_S$.

We can split the run ρ into segments

$$A_1, B_1, A_2 \ldots, B_{k-1}, A_k$$

such that:

- Every A_i is a sequence of states (sub-run) where code of S is executed (formally transitions within A_k are labeled by names of classes from E). Moreover, every A_i except for the last one ends with a state of the form $(e_i[C_i.m_i(\vec{a}_i)], s_i)$ where the subexpression $C_i.m_i(\vec{a}_i)$ is about to be rewritten (with C_i defined in E). We will denote the tuple $(C_i, m_i, \vec{a}_i)$ by x_i.

$$P \vdash \langle \texttt{Cast}\ C\ (\texttt{throw}\ e), s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{45}$$

$$P \vdash \langle V := \texttt{throw}\ e, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{46}$$

$$P \vdash \langle \texttt{throw}\ e.F\{D\}, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{47}$$

$$P \vdash \langle \texttt{throw}\ e.F\{D\} := e_2, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{48}$$

$$P \vdash \langle \texttt{Val}\ v.F\{D\} := \texttt{throw}\ e, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{49}$$

$$P \vdash \langle \texttt{throw}\ e\ \ll bop \gg\ e_2, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{50}$$

$$P \vdash \langle \texttt{Val}\ v_1\ \ll bop \gg\ \texttt{throw}\ e, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{51}$$

$$P \vdash \langle \{V : T; \texttt{Throw}\ a\}_D, s\rangle \xrightarrow{D} \langle \texttt{Throw}\ a, s\rangle \tag{52}$$

$$P \vdash \langle \{V : T; V := \texttt{Val}\ v;\ \texttt{Throw}\ a\}_D, s\rangle \xrightarrow{D} \langle \texttt{Throw}\ a, s\rangle \tag{53}$$

$$P \vdash \langle \texttt{throw}\ e.M(es), s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{54}$$

$$P \vdash \langle \texttt{Val}\ v.M(map\ \texttt{Val}\ vs\ @\ (\texttt{throw}\ e\ \cdot\ es')), s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{55}$$

$$P \vdash \langle \texttt{throw}\ e;\ e_2, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{56}$$

$$P \vdash \langle \texttt{if}(\texttt{throw}\ e)\ e_1\ \texttt{else}\ e_2, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{57}$$

$$P \vdash \langle \texttt{throw}(\texttt{throw}\ e), s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{58}$$

Figure 8. Exception propagation

$$\frac{P \vdash \langle es, s\rangle\ [\xrightarrow{\ell}]\ \langle es', s'\rangle}{P \vdash \langle D.M(es), s\rangle \xrightarrow{\ell} \langle D.M(es'), s'\rangle} \tag{59}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle e[e_2],\ s\rangle \xrightarrow{\ell} \langle e'[e_2],\ s'\rangle} \tag{60}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle (\texttt{Val}\ v)[e],\ s\rangle \xrightarrow{\ell} \langle (\texttt{Val}\ v)[e'],\ s'\rangle} \tag{61}$$

$$P \vdash \langle D.M(map\ \texttt{Val}\ vs\ @\ (\texttt{throw}\ e\ \cdot\ es')), s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{62}$$

$$P \vdash \langle (\texttt{throw}\ e)[e'], s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{63}$$

$$P \vdash \langle e'[\texttt{throw}\ e], s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{64}$$

Figure 9. Subexpression reduction and exception propagation rules for Jinja+.

$$\frac{P \vdash\ D\ \textit{has-static}\ M:\ Ts \to T = (pns,\ body) \qquad |vs| = |pns| \qquad |Ts| = |pns|}{P \vdash \langle D.M(map\ \texttt{Val}\ vs), s\rangle \mathrel{\bar{\to}} \langle blocks_D(pns, Ts, vs,\ body), s\rangle} \tag{65}$$

$$\frac{n \geq 0,\ \textit{new-Addr}\ h = a}{P \vdash \langle \texttt{new}\ \tau\texttt{[}intg(n)\texttt{]},\ (h,l)\rangle \to \langle addr\ a, (h(a \mapsto initArr(\tau,n)), l)\rangle} \tag{66}$$

$$P \vdash \langle null.F\{D\}, s\rangle \mathrel{\bar{\to}} \langle \texttt{THROW NullPointerException},\ s\rangle \tag{67}$$

$$\frac{n < 0}{P \vdash \langle \texttt{new}\ \tau\texttt{[}intg(n)\texttt{]},\ (h,l)\rangle \to \langle \texttt{THROW NegativeArraySizeException},\ (h,l)\rangle} \tag{68}$$

$$\frac{h\ a = (\tau, m, t),\ 0 \leq n < m,\ t(n) = v}{P \vdash \langle (addr\ a)\texttt{[}intg\ n\texttt{]},\ (h,l)\rangle \to \langle Val\ v,\ (h,l)\rangle} \tag{69}$$

$$\frac{h\ a = (\tau, m, t),\ \neg(0 \leq n < m),}{P \vdash \langle (addr\ a)\texttt{[}intg\ n\texttt{]},\ (h,l)\rangle \to \langle \texttt{THROW IndexOutOfBoundsException},\ (h,l)\rangle} \tag{70}$$

$$\frac{h\ a = (\tau, m, t),}{P \vdash \langle (addr\ a)\texttt{.lenght},\ (h,l)\rangle \to \langle intg\ m,\ (h,l)\rangle} \tag{71}$$

$$\frac{h\ a = (\tau, m, t),\ 0 \leq n < m,\ isOfType(v,\tau),\ t' = arrayUpdate(t,n,v)}{P \vdash \langle (addr\ a)\texttt{[}intg\ n\texttt{] :=}\ Val\ v,\ (h,l)\rangle \to \langle Unit,\ (h(a \mapsto (\tau,m,t')), l)\rangle} \tag{72}$$

$$\frac{h\ a = (\tau, m, t),\ \neg(0 \leq n < m),}{P \vdash \langle (addr\ a)\texttt{[}intg\ n\texttt{] :=}\ Val\ v,\ (h,l)\rangle \to \langle \texttt{THROW IndexOutOfBoundsException},\ (h,l)\rangle} \tag{73}$$

$$\frac{h\ a = (\tau, m, t),\ 0 \leq n < m,\ \neg isOfType(v,\tau),}{P \vdash \langle (addr\ a)\texttt{[}intg\ n\texttt{] :=}\ Val\ v,\ (h,l)\rangle \to \langle \texttt{THROW ArrayStoreException},\ (h,l)\rangle} \tag{74}$$

Figure 10. (Exceptional) expression reduction rules for Jinja+, where: Function $initArr(\tau,n)$ returns an array of length n with elements initialized to the default value of type τ. Expression $P \vdash D$ *has-static* $M : Ts \to T = (pbs, body)$ means that in program P, class D contains declaration of static method M with argument types Ts, return type T, formal arguments pbs, and the body $body$.

– Every B_i is a sequence of states where code of E is executed. It begins with $(e_i[\{e'_i\}_{C_i}], s_i)$, where $\{e'_i\}_{C_i}$ is the block obtained by the static method call rule applied to $C_i.m_i(\vec{a}_i)$ (it depends only on C_i, m_i, and $\vec{a}_i$), and ends with $(e_i[\{y_i\}_{C_i}], t_i)$, where y_i is a value (that is return by this method).

We will represent such a run as

$$\rho = A_1[s_1, x_1]B_1[t_1, y_1]A_2[s_2, x_2]\ldots B_{k-1}[t_{k-1}, y_{k-1}]A_k$$

The square brackets, intuitively, contain all the information that is passed between S and E. We will write A_i^ρ to denote A_i, and similarly for A_i, B_i, x_i, y_i, s_i, and t_i when it is necessary to make it explicit which run we consider.

The following result states, so called, state separation of E and S: for a representation of a run as above, B_i does not change the part of the state that can be reached from S and, similarly, A_i does not change the part of the state that can be reached by E.

Lemma 4. *We have:*

1) $s_i|_S = t_i|_S$,

2) $t_i|_E = s_{i+1}|_E$.

Intuitively, this lemma holds true, because E and S do not exchange references (they exchange only primitive values) and do not share any static variables.

Because of this state separation, we can obtain the following two results. The first one states that the state of E (the part of the state that E can access) and the values that E returns depend solely on the input it explicitly gets from S by method calls (recall that the values passed by such calls are, by our assumption, of primitive types only).

Lemma 5. *Let S, S' and E be like in the theorem. Let ρ be the run of $E \cdot S$ and ρ' be the run of $E \cdot S'$. If*

$$x_1^\rho = x_1^{\rho'}, \ldots, x_k^\rho = x_k^{\rho'},$$

then

$$t_k^\rho|_E \sim t_k^{\rho'}|_E \quad \textit{and} \quad y_k^\rho = y_k^{\rho'}.$$

Conversely, the state of S and the values it provides to E, solely depend on the values that E returns to S:

Lemma 6. *Let S, E and E' be like in the theorem. Let ρ be the run of $E \cdot S$ and ρ' be the run of $E' \cdot S$. If*

$$y_1^\rho = y_1^{\rho'}, \dots, y_k^\rho = y_k^{\rho'},$$

then

$$s_{k+1}^\rho|_S \sim s_{k+1}^{\rho'}|_S \quad \textit{and} \quad x_{k+1}^\rho = x_{k+1}^{\rho'}$$

Proof of Theorem 5. Implication from left to right is obvious. So let us assume that, I-noninterference *does not hold* for S. It means that there exists an I-environment E for S such that noninterference does not hold for $E \cdot S$. This, in turn, means that there are valid $\vec{a}_1$ and $\vec{a}_2$ such that the run ρ of $E \cdot S[a_1]$ and the run ρ' of $E \cdot S[a_1]$ give different results (i.e. both runs are finite and the final value of `result` is different).

In the following, we only consider the case where the number of blocks B_i in both runs is the same (the other case can be handled in a similar way).

As the value of `result` in a state s is part of $s|_E$, we conclude from Lemma 5, that there exists an index k such that $x_k^\rho \neq x_k^{\rho'}$. Let k be the first such index. Note that $y_i^\rho = y_i^{\rho'}$ for $i \in \{1, \dots, k-1\}$. Let us assume that the first argument in the call described by x_k^ρ has value z which is different than the value z' of the first argument in $x_k^{\rho'}$ (for the other cases the proof is very similar).

We define now a sequence $\vec{u}$ as the sequence containing only zeros with the following exceptions:

– In the system $\tilde{E}_{\vec{u}} \cdot S[a_i]$, the consecutive $(k-1)$ values that methods of $\tilde{E}_{\vec{u}}$ return to $S[a_i]$ are determined by some subsequence $u_{p_1}, \dots, u_{p_k}$. Therefore we set $u_{p_i} = y_i$ for $i \in \{1, \dots, k-1\}$ (that is, the values returned in $\tilde{E}_{\vec{u}} \cdot S[a_i]$ coincide with the values returned in $E \cdot S[a_i]$).

– Let l be the integer such that u_l decides whether to test the first arguments in B_k-th block (then this argument is compared to u_{l+1} to determine the result). We set u_l to 1 and u_{l+1} to z (as defined above). Note that $l \notin \{p_1, \dots, p_{k-1}\}$.

– As mentioned, for all remaining i we set $u_i = 0$.

Now, to complete the proof it is enough to show that $\tilde{E}_{\vec{u}} \cdot S[a_1]$ and $\tilde{E}_{\vec{u}} \cdot S[a_2]$ give different results.

Let σ be the run of the system $\tilde{E}_{\vec{u}} \cdot S[a_1]$ and σ' be the run of the system $\tilde{E}_{\vec{u}} \cdot S[a_2]$. As, by the definition of $\vec{u}$, we know that $y_i^\rho = y_i^{\rho'} = y_i^\sigma = y_i^{\sigma'}$ for $i \in \{1, \dots, k-1\}$, we can use Lemma 6 to obtain $x_k^\sigma = x_k^\rho$ and $x_k^{\sigma'} = x_k^{\rho'}$. Therefore, the first argument in x_k^σ is z and it is different that the first argument in $x_k^{\sigma'}$. Therefore, by the definition of $\vec{u}$ (more precisely, by the values of u_l and u_{l+1} the variable `result` is set to `true` in $\tilde{E}_{\vec{u}} \cdot S[a_1]$ and to `false` in $\tilde{E}_{\vec{u}} \cdot S[a_2]$, after which both systems terminate (the `abort` is executed immediately after the assignment). Hence, these systems give different results, which completes the proof.

B. Communication through Arrays, Simple Objects, and Exceptions

The proof of Theorem 6 is very similar—and only slightly more complicated–to the proof of Theorem 5. Indeed, the assumptions we have taken about the program S guarantee that—even if, technically, some reference are exchanged between E and S—the communication between E and S is, effectively, as if only pure values were exchanged.

Because of that we only mention those points in the proof which differ from the analogous points in the proof of Theorem 5.

1. There are the following changes in the representation of a run: (a) x_i now records not only primitive values of arguments, but also *values* of arrays of bytes (not the references to these arrays though); (b) y_i may now be a primitive value, a *value* of an array (but not the reference), or a *value* of a simple object, that is a collection of all values of the fields of a returned object that are in I_E.
2. Item 1) of Lemma 4 needs to be changed: $t_i|_S$ can now differ from $s_i|_S$, but only on the values in the arrays that S has passed to E (as references to these arrays are not used by S anymore, this will not cause any problem in the results that follow).

With the above changes in the definition of representation (item 1. above) Lemma 5 and Lemma 6 stay true also for S and E like in Theorem 6. Now, these facts can be used to prove Theorem 6 in a very similar way as in the case of Theorem 5.

APPENDIX E. PROOF OF THEOREM 7

Before we give the proof of Theorem 7, we want to point out some critical points and assumptions that are used in this proof.

1. First, we assume that we have a system CCA2Enc providing the interface I_{Enc} and such that it correctly implements CCA2-secure public encryption scheme (we do not prove correctness of this implementation). We assume that, in particular, the above mentioned implementation does not fail (i.e. always returns the expected result) unless the expected result is too big to fit within an array (recall that the maximum size of an array depends on the security parameter and the function *intsize*).
   We also assume that this encryption scheme is such that the length of a ciphertext is the (polynomially computable) function of the length of the encrypted plaintext and vice versa.
2. It is critical to assume that the Jinja program has unbounded memory, as otherwise the asymptotic notion of security our results are based upon does not make sense.

Now, we shortly present and ideal functionality $\mathscr{F}$ and a real functionality $\mathscr{R}$ for public key encryption in the Turing machine model following [25]

TM functionalities: Different instances of functionalities are distinguished by different ID-s, sent with each request. The functionalities accept the following requests (where the request is written on the input tape of a TM)

1. *Initialization-Decryptor*: The functionality is supposed to return a public key.
2. *Initialization-Encryptor* The functionality responds with an *comleted* message.
3. *Encryption(pk,m)*: The functionality is supposed to encrypt m with pk and return the result.
4. *Decryption(m)*: The functionality is supposed to decrypt m using the stored private key.

Both the real and the ideal functionality, on initialization, obtain a corruption bit. Because, in our simulation, the environment never corrupts functionalities, we will skip the description of actions of these functionalities if this bit is set to 1.

The real functionality $\mathscr{R}$, on initialization (be it *Initialization-Decryptor* or *Initialization-Encryptor*), generates a fresh public/private key pair and returns the public key. Then, it uses the private key to decrypt message, and the key pk provided in the encrypt request to encrypt messages.

The ideal functionality, on creation, asks the environment (the simulator) for a crypto algorithm it will use, and creates a key pair using one of the provided algorithms. On encrypt requests, it, similarly to the considered Jinja+ functionality, encrypts an unrelated message of the same length (provided the functionality is not corrupted) and stores it along with the plaintext. On decryption, if the key pk is the same as the public key stored in the functionality, it tries to retrieve the corresponding plaintext from the table (as the $IdealPKE$ does). Otherwise it uses the provided decryption algorithm to decrypt the message.

See [25] for details.

We want to prove that $\mathsf{RealPKE}$ realizes $\mathsf{IdealPKE}$ w.r.t. I_{PKE}. In this proof we will use a result from [25] that $\mathscr{R}$ realizes $\mathscr{F}$. Let $\mathscr{S}$ be the simulator used in the realization proof in [25]. The simulator for $\mathsf{IdealPKE}$ we will use in the proof is $S = \mathsf{CCA2Enc}$, as described above.

Let E be a bounded-environment with $I_{PKE} \vdash E$.

Simulating E: We define a Turing machine M_E that simulates E. Clearly, every complete Jinja+ program can be simulated by Turing machine. Moreover, if a program is bounded (for a given *intsize*), then its simulation is also polynomial (recall that a run with security parameter η uses integers of maximal size $intsize(\eta)$; operations on integers of this size can be polynomially simulated by a Turing machine).

In our case, however, the system E we consider is not a complete Jinja+ program; it interacts with another system (such as $\mathsf{RealPKE}$ or $\mathsf{IdealPKE}$). Therefore we assume that M_E communicates with another Turing machine (or more generally, a system of Turing machines).

The machine M_E is defined in such a way that it maintains a representation of a Jinja+ state, the state of E. In this representation, references are represented by consecutive identifiers. We distinguish two types of references: those pointing to an *internal* object, that is instances of a classes defined in E or an arrays, and those pointing to an *external* object which can be either instances of Encryptor of Decryptor. For each reference to an internal object, a representation of this object is remembered by M_E. For references to external objects this is not the case (some additional information, however, can be stored along with these references; see below). A method call for an internal reference is modeled internally in M_E; a method call to an external object is realized by triggering another Turing machine.

When the simulation of E by M_E if finished, this machine outputs the value of the (simulated) variable `result`.

Method invocations for external references are simulated in the following way:

1. **Creating a new instance of `Decryptor`:** M_E creates a new instance of *Decryptor* (Turing machine) by sending the *Initialization-Decryptor* request with a fresh identifier id. This identifier will be used as the reference to this object. M_E waits then for a response with a public key. This key is stored together with id.
2. **`Decryptor.getEncryptor` for an object represented by id:** M_E creates a new instance of *Encryptor* (TM) by sending the *Initialization-Encryptor* request with id and a fresh identifier id', which will serve as the reference to this object. The identifier id' is stored together with id .
3. **`Decryptor.decrypt` for an object represented by id and array r:** M_E sends *Encryption* request to machine id with the data stored under r, and waits for the response. A response is a sequence of bytes. M_E simulates creation of a new array and copies the obtained byte-sting to this array.
4. **`Encryptor.encrypt` for an object represented by id' and array r:** M_E retrieves (the decryptor id and) the public key associated with id' and uses it in the request *Encrypt* along with id, id' and the data stored under r. A response is a sequence of bytes. M_E simulates creation of a new array and copies the obtained byte-sting to this array.
5. **`Encryptor.getPublicKey` for an object represented by id':** M_E retrieves the public key associated with the encryptor (without any external call).

Representing runs: Let T be either the system $\mathsf{RealPKE}$ or the system $(S \cdot \mathsf{IdealPKE})$. Let u be a random input (a sequence of bits) and η be a security parameter. Like in Appendix D-A, the (deterministic) finite run ρ of $E \cdot T$ with random input u and security parameter η can be represented as

$$A_1[s_1, x_1]B_1[t_1, y_1]A_2 \cdots B_{n-1}[t_{n-1}, y_{n-1}]A_n[s_n]$$

where

- Every A_i is a part of the run (a sequence of configurations) where only expressions originating from E are reduced, i.e. all the transitions in A_i are labelled with names of classes defined in E. Every A_i, except for the last one, ends with a state of the form $(e_i[e'_i], s_i)$ where the subexpression e'_i is about to be rewritten by a method invocation rule.
- Every B_i is a part of run where only expressions originating from T are reduced. It begins with $(e_i[\{e''_i\}_D], s_i)$, where $\{e''_i\}_D$ is the block obtained by applying the method invocation rule to e'_i for some class D defined in T (it depends only on e'_i), and ends with $(e_i[\{v_i\}_D], \bar{s}_i)$, where v is a value (that is return by the method).
- s_i and t_i are the states after A_i and B_i, respectively.
- By x_i we denote the *invocation data* consisting of the name of the called method and the values passed as arguments (if an argument is of type `byte[]` then x_i contains the values in the array, not the reference to this array). This data is determined by e_i and s_i.
- By y_i we denote the return value (again, if an array is returned, then y_i contains the values in this array, not the reference). This return value is determined by v_i and t_i.

Similarly, we represent the (deterministic) execution $\tilde{\rho}$ of the system of Turing machines $M_E|M_T$ with random input u and security parameter η, where M_E is defined above and M_T is either $\mathscr{R}$ or $(\mathscr{S}|\mathscr{F})$ as

$$\tilde{A}_1[\tilde{s}_1, \tilde{x}_1]\tilde{B}_1[\tilde{t}_1, \tilde{y}_1]\tilde{A}_2 \cdots \tilde{B}_{n-1}[\tilde{t}_{n-1}, \tilde{y}_{n-1}]\tilde{A}_n[s_n]$$

where

- Every $\tilde{A}_i$ is a part of the run of the system where M_E is active. Every $\tilde{A}_i$, except for the last one, ends with M_E sending data $\tilde{x}_i$ to M_T (and activating M_T).
- Every $\tilde{B}_i$ is a part of the run of the system where M_T is active. It ends with M_T sending a response $\tilde{y}_i$ back to M_E.
- $\tilde{s}_i$ is the state of M_E after $\tilde{A}_i$ (notice the difference to s_i which was the state of the whole system after A_i).
- $\tilde{t}_i$ is the state of M_T after $\tilde{B}_i$ (notice, as above, the difference to t_i).

Let $s = (h, l)$ be a Jinja+ state that occurs in the run ρ of $E \cdot T$. We want to define the part of the state s that "belongs" to E and the part that "belongs" to T.

We define $h|_E$ to be the restriction of h to only those references that, in the run ρ, have been created by E or have been obtained by E as a return value from a call to T. By $h|_T$ we denote the restriction of s to the remaining references, that is the references in the run ρ that have been created by T but not returned to E.

We define $l|_E$ to be the restriction of l to those (static) variables that are accessible from E. Similarly, $l|_T$ denotes the restriction of l to those static variables that are accessible from T. Note that these restrictions are disjoint except for the read-only security parameter (T does not access any static fields of E; E does not access any static fields of T).

We take $s|_E = (h|_E, l|_E)$ and $s|_T = (h|_T, l|_T)$.

Let $\tilde{s}$ be a Jinja state as represented by M_E and s be a (real) Jinja state. We say that $\tilde{s}$ represents $s = (h, l)$, written $\tilde{s} \models s$, if there is a function f from identifiers (that represent references in M_E) to (Jinja) references (addresses) such that

- the domain of h is $f(X)$ where X is the set of identifiers used by M_E to represent references,
- if $\tilde{r} \in X$, then the representation of the object pointed by $\tilde{r}$ agrees with the object pointed by r (in the Jinja state) in the following sense: (i) corresponding fields (in the TM representation and in the Jinja object) of primitive types have the same values, (ii) if a filed of the TM representation contains an identifier id, then the corresponding field of the Jinja object contains $f(id)$.
- The values of variables in l are—up to mapping f—the same as the values in the TM representation of l.

We say that $\tilde{x}_i$ matches x_i, where $\tilde{x}_i$ and x_i are as above, if the requests $\tilde{x}_i$ is the translation of the method invocation x_i, as specified in the simulation process above. In a similar way, we can say that a response $\tilde{y}_i$ matches y_i.

Relation between Jinja runs and TM runs: Now we are ready to relate the runs of the corresponding Jinja programs and Turing machine systems, as introduced above.

Lemma 7. *For every random input u and every security parameter η (and $A_i, \tilde{A}_i, \ldots$ as above) we have:*
(a) $s_i|_E = t_i|_E$ and $t_i|_T = s_{i+1}|_T$,
(b) $\tilde{x}_i$ matches x_i,
(c) $\tilde{y}_i$ matches y_i,
(d) $\tilde{s}_i \models s_i|_E$,
(e) $\tilde{t}_i \models t_i|_T$,

Item (a) states that sub-states of E and T are separated (the execution of A_i does not changes what T can access and the execution of B_i does not change what E can access).

Items (b) and (c) state that the components E and T in the Jinja run and the corresponding components in the TM system exchange exactly the same data, up to the provided translation.

Item (d) states that M_E correctly simulates E (which is given by the definition of M_E).

Item (e) states that the Jinja program T is functionally equivalent to the corresponding Turing machine M_T. In particular, for the same input, $\mathscr{R}$ produces the same data as RealPKE and $\mathscr{S}|\mathscr{F}$ produces the same data as $S \cdot$IdealPKE. This is given by the definition of these systems.

In the reasoning below, we leverage the fact that, without loss of generality, we can assume that E, when connected with T, never makes requests to T that fail (i.e. never makes method calls that return `null`). This is because E can compute the expected size of the output message (recall that we assumed that the length of a plaintext and a corresponding ciphertext are polynomially related). Therefore E can predict potential failure and avoid requests that would fail (E does not lose any information by not executing these requests, as it knows the result up front).

Now, we can observe that a direct consequence of the above lemma (more precisely, of the fact that $\tilde{s}_n \models s_n|_E$) is that the final value of variable `result` in ρ and $\tilde{\rho}$ is the same and, therefore, these (finite) runs output the same result. As it holds for all random input u and all security parameters η, up to some negligible function, the system $E \cdot \mathsf{RealPKE}$ outputs true with the same probability the system $M_E|\mathscr{R}$ outputs 1 and the system $E \cdot S \cdot \mathsf{IdealPKE}$ outputs true with exactly the same probability the system $M_E|\mathscr{S}|\mathscr{F}$ outputs 1. Now, as we know that $M_E|\mathscr{R} \equiv M_E|\mathscr{S}|\mathscr{F}$, it follows that the probability that true is output by $E \cdot \mathsf{RealPKE}$ and by $E \cdot S \cdot \mathsf{IdealPKE}$ is the same up to some negligible value.

APPENDIX F. VERIFICATION USING JOANA

Program slicing is a well known static analysis technique that is used to capture dependencies between program statements. It can be applied to check a program for sequential termination-insensitive non-interference. However, when the program changes, the analysis has to be redone in order to guarantee non-interference. In this subsection we sketch a proof that under certain restrictions small changes to the program do not change the analysis result with regard to information flow and noninterference. The technique proposed here allows Joana to verify families of programs considered in Section VII.

A. Introduction to Information Flow Control with Slicing

The concept of program slicing was first introduced by Weiser [39], where he argued about how a programmer tries to determine the relevant program parts responsible for a specific error: He identifies the program statement s where the error occurred and subsequently infers all statements that may have influenced the execution of s. These statements are called the *backward slice* of s. Later, Ottenstein and Ottenstein [33] proposed using *program dependence graphs* (PDG) in order to compute program slices. PDGs are a graph based representation of the program semantics, where a node corresponds to a statement and edges correspond to dependencies between statements.

The Joana project [19], [18] applies a very precise slicer to information flow control for Java programs. It has been developed and improved for several years and can analyze medium sized programs up to 80.000 LoC. Joana takes a set of secret (high) input statements and a set of untrusted (low) output statements as arguments and checks if an illegal flow from high to low is possible. Therefore it computes the PDG representation from the program source and marks the PDG nodes that correspond to high or low statements with matching security labels. Then a dataflow analysis[2] on the PDG propagates the input security labels along the graph edges. Finally Joana checks if any high security label has reached a statement labeled as low output. If no information leaking low statements are found, the program is guaranteed to be secure.

In case of a security violation slicing is used to determine the high input statements involved: All high input statements included in the backward slice of an information leaking low statement are potentially leaked. So whenever information flows from a high statement h to a low statement l, h must be included in the backward slice of l. If l is not reachable from h then the absence of information flow is guaranteed.

B. PDG Structure and Computation

```
class Node {
    int value;
    Node next;
    Node(int v, Node n) {
        value = v;  next = n;
    }
}
private static Node list = null;
private static boolean listInitialized = false;
private static Node initialValue() {
    /* WILL BE MODIFIED */
    return new Node(1, new Node(2, null)); // V2
//  return new Node(1, null);              // V1
}
static public int untrustedInput() {
    if (!listInitialized) {
        list = initialValue();
        listInitialized = true;
    }
    if (list==null) return 0;
    int tmp = list.value;
    list = list.next;
    return tmp;
}
```

Figure 11. Code that models unknown input in form of a linked list as proposed in section VII.

```
public class P {
  public static void main(String[] argv) {
        int untrusted1 = untrustedInput();
        int secret = 42;
        int unstrusted2 = untrustedInput();
        low = unstrusted1;
        secret += 21;
        low += unstrusted2;
  }
}
```

Figure 12. An example of a program without illegal information flow using untrusted input from Figure 11.

Figure 13 shows a PDG for the example program in Figure 11 and 12. It contains a node for each statement

[2]As Joana supports declassification and any kind of security labels that form a lattice, a simple reachability analysis is not enough in general.

and additional *entry nodes* (bold outline) that correspond to method entries. Entry nodes have special fields *IN* and *OUT* representing the input and output of the method. Nodes of method call statements also include those fields representing the values sent to and received from the called method. The nodes of each single method form a subgraph that is only connected to other methods through edges between calls and entry nodes. We distinguish four kinds of edges through which information flow may occur:

– *control dependence edge*: Statement n is control dependent on m iff the outcome of the execution of m decides if n is executed. E.g. all statements in the body of an if-clause are control dependent on the condition statement. Control dependencies are computed for each method through an intraprocedural analysis on its *control flow graph*. They capture implicit flow of information.
– *data dependence edge*: Statement n is data dependent on statement m iff n reads a value that m has produced. This includes the use and definition of variables as well as referenced and modified values on the heap. Data dependencies on heap values are computed with the help of a *points-to analysis*. A points-to analysis computes for each heap reference at which locations on the heap it may point to. When statement m modifies the same location statement n reads and n is executed after m, n is data dependent on m. Data dependencies connect only statements that belong to the same method. In order to capture flow between methods, call and parameter edges are used.
– *call edge*: A call edge connects a method call statement to the entry node of the called method.
– *parameter and heap dependence edge*: Parameter and heap dependence edges capture the flow of information from a method call to the called method and back. They include the parameters passed into the method and the return value passed back to the callsite and also heap locations referenced and modified by the method.

C. Problem: Parameterized initialization

We consider programs $P_{\vec{u}}$ consisting two parts: P_{main} and $E_{\vec{u}}$, where P_{main} is a system that meets the conditions given below and $E_{\vec{u}}$ is as in Figure 11, for the variant of the method `initialValue`$_{\vec{u}_0}$ determined by a sequence $\vec{u}$, as in Section VII. We require (which is consistent with assumption of Section VII) that

– P_{main} does not reference nor modify the variable `list`,
– P_{main} does not create object of class `Node`,

Note that with these conditions the following statements are true:

– Only `untrustedInput` references and modifies `list`.
– A single static method `initialValue`$_{\vec{u}_0}$ creates all list elements and initializes the variable `list`.
– The constructor for elements of `list` is only called inside `initialValue`$_{\vec{u}_0}$.
– `initialValue`$_{\vec{u}_0}$ has no parameters and does not reference or modify any static variable.
– neither `untrustedInput` nor `initialValue`$_{\vec{u}_0}$ contain statements that reference high (secret) or low variables.

Using these properties of programs P as described above, we can prove the following result.

Proposition 1. For all P_{main} such that $P_{\vec{u}} = P_{main} \cdot E_{\vec{u}}$ comply with the requirements stated above, if $P_{\vec{u}_0}$ does not contain illegal information flow, for some sequence $\vec{u}_0$, then $P_{\vec{u}}$ does not contain illegal information flow for all $\vec{u}$.

Proposition 1 enables us to reason about a family of programs by analysing only a singe instance of it. We take some specification of $\vec{u}_0$ and analyze $P_{\vec{u}_0}$ If the analysis guarantees security in this case, we know due to Proposition 1 that the program is secure for all versions of `initialValue`$_{\vec{u}_0}$ fulfilling the restrictions.

Because programs $T_{\vec{u}}$ we consider in Section IX-B comply with the assumptions of the above theorem, we can show noninterference of this family of programs by showing noninterference of only one specific instance. Indeed, we have used Joana to verify that such an instance (and hence the whole family of programs) is indeed secure.

D. Example

We can take the program from Figure 12 as P_{main}. Indeed all the above requirements are met for $P_{\vec{u}} = P_{main} \cdot E_{\vec{u}}$.

This program includes reading and writing of untrusted values as well as a computation on a secret value. Figure 13 contains the corresponding PDG for $\vec{u} = 1, 2$. This program is considered secure, because the PDG contains no connection from the secret value in line 28 to a statement that leaks to untrusted output (line 30 and 32).

E. Proof sketch of Proposition 1

Let us assume that the program $P_{\vec{u}_0} = P_{main} \cdot E_{\vec{u}_0}$, for some $\vec{u}_0$ is secure. Note that we have assumed a set of statements $S_{secret} \subseteq P_{main}$ that refers to secret values in $P_{\vec{u}_0}$ and a set of statements $S_{out} \subseteq P_{main}$ that refer to untrusted output (e.g. changing the value of a low variable). As $P_{\vec{u}_0}$ is guaranteed to be secure, we also know that there is no path in the corresponding $PDG_{P_{\vec{u}_0}}$ from secret values to untrusted output:

$$\forall l \in S_{out} : \text{slice}_{PDG_{P_{\vec{u}_0}}}(l) \cap S_{secret} = \{\}$$

If we use $Path(PDG_{P_{\vec{u}_0}})$ as the set of all possible paths in $PDG_{P_{\vec{u}_0}}$ we get:

$$\forall h \in S_{secret}\ \forall l \in S_{out}\ \forall p \in Path(PDG_{P_{\vec{u}_0}}) : h \in p \implies l \notin p_{|h}$$

Let

$$p_{|h} = (p_1 \to \ldots \to h \to p_i \to \ldots \to p_n)_{|h} = (h \to p_i \to \ldots \to p_n)$$

We will use the following, auxiliary lemma:

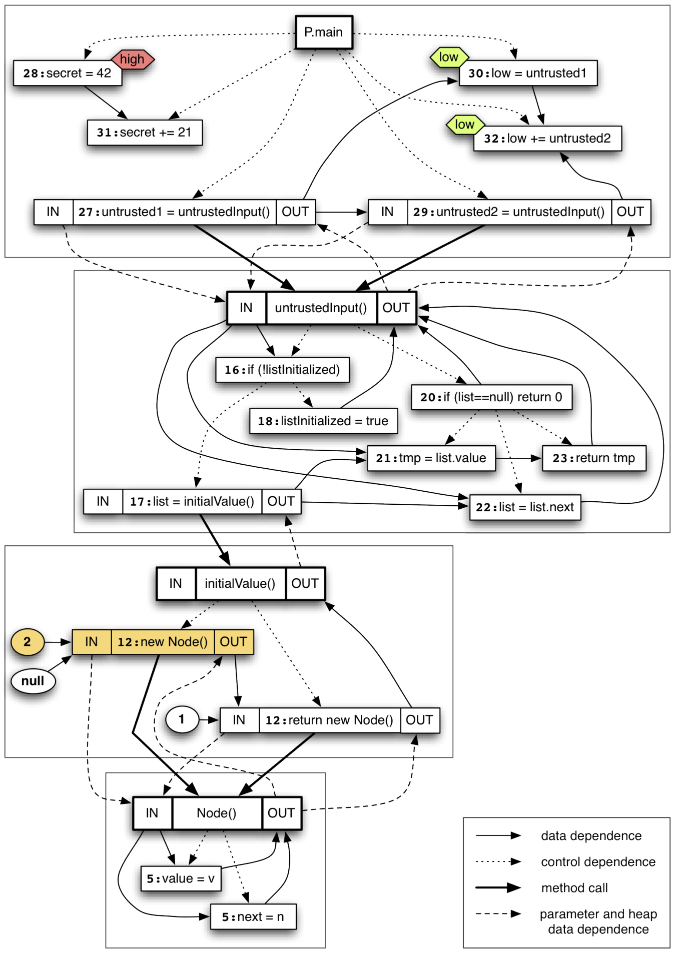

Figure 13. A PDG of the example program from Figure 11 and 12. The colored nodes are the additional nodes that occur if version $V2$ of $\texttt{initialValue}_{\vec{u}_0}$ is chosen instead of $V1$.

Lemma 8. *Changes in the PDG local to* $\texttt{initialValue}_{\vec{u}_0}$ *are not affecting the security guarantee.*

Proof: If the PDG local to $\texttt{initialValue}_{\vec{u}_0}$ has changed, either previously existing nodes or edges were removed or new ones were introduced. Removing nodes or edges has no effect on the security guarantee as it only reduces the number of possible paths in the whole PDG. So no additional flow can be introduced. Additional nodes or edges inside the PDG part of $\texttt{initialValue}_{\vec{u}_0}$ does increase the number of possible paths inside the method, but not the number of paths leading to or leaving from it. Because $\texttt{initialValue}_{\vec{u}_0} \cap S_{secret} = \{\}$ and $\texttt{initialValue}_{\vec{u}_0} \cap S_{out} = \{\}$ we get that no illegal flow can not start or end inside of $\texttt{initialValue}_{\vec{u}_0}$. Due to the restrictions we know that $\texttt{initialValue}_{\vec{u}_0}$ does not take any parameters as arguments and that it does not reference any static variables. So no parameter and heap dependence edges are connected to the entry node of $\texttt{initialValue}_{\vec{u}_0}$. The only possible paths from a secret source to an untrusted output through $\texttt{initialValue}_{\vec{u}_0}$ have to enter through the call edge and to leave through the parameter and heap dependence of the *OUT* field. Because any output of a method is dependent on the method execution, there is always a path from the entry node to its *OUT* field. So additional nodes or edges inside $\texttt{initialValue}_{\vec{u}_0}$ do not introduce new flow from S_{secret} to S_{out}. ■

Now we change $\texttt{initialValue}_{\vec{u}_0}$ of $P_{\vec{u}_0}$ to $\texttt{initialValue}_{\vec{u}}$, given by some $\vec{u}$, and assume that the resulting program $P_{\vec{u}} = P_{main} \cup E_{\vec{u}}$ contains an illegal flow. Then we show by contradiction that this is not possible and thus $P_{\vec{u}}$ must be secure.

By the above assumption

$$\exists h \in S_{secret}\ \exists l \in S_{out}\ : h \in \text{slice}_{PDG_{P_{\vec{u}}}}(l)$$

which means there must be a path p from h to l in the changed PDG that was not part of the original.

$$\exists p \in Path(PDG_{P_{\vec{u}}}) : h \in p \land l \in p_{|h} \land p \notin Path(PDG_{P_{\vec{u}_0}})$$

Path p is a list of instructions that are connected through dependencies. We are now going to show that any of these dependencies must already have been part of $PDG_{P_{\vec{u}_0}}$.

If $p \notin Path(PDG_{P_{\vec{u}_0}})$ then p must contain at least one edge $n_1 \longrightarrow n_2$ that is not in $PDG_{P_{\vec{u}_0}}$.

$$p \notin Path(PDG_{P_{\vec{u}_0}}) \implies \exists n_1 \longrightarrow n_2 \in p \mid n_1 \longrightarrow n_2 \notin PDG_{P_{\vec{u}_0}}$$

Now we consider four different cases.

Case $n_1 \notin PDG_{P_{\vec{u}_0}}, n_2 \notin PDG_{P_{\vec{u}_0}}$**.** Both nodes have not been part of $PDG_{P_{\vec{u}_0}}$. Because $\texttt{initialValue}_{\vec{u}}$ is the only changed part of $P_{\vec{u}}$ we know that $n_1, n_2 \in \texttt{initialValue}_{\vec{u}}$. Due to Lemma 8 this edge can not introduce new illegal flow.

Case $n_1 \notin PDG_{P_{\vec{u}_0}}, n_2 \in PDG_{P_{\vec{u}_0}}$**.** Because $\texttt{initialValue}_{\vec{u}}$ is the only changed part of $P_{\vec{u}}$ we know that $n_1 \in \texttt{initialValue}_{\vec{u}}$. So due to Lemma 8 this edge cannot introduce new illegal flow.

Case $n_1 \in PDG_{P_{\vec{u}_0}}, n_2 \notin PDG_{P_{\vec{u}_0}}$**.** $n_2 \in \texttt{initialValue}_{\vec{u}}$ so no new illegal flow is introduced with this edge.

Case $n_1, n_2 \in PDG_{P_{\vec{u}_0}}$**.** Only the edge between n_1 and n_2 is new in $PDG_{P_{\vec{u}}}$. Because of Lemma 8 we only have to consider the case $n_1 \longrightarrow n_2 \notin \texttt{initialValue}_{\vec{u}}$. The new edge may be of four different kinds:

– *control dependence edge*: Control dependencies are computed per method using the control flow graph. No method in $P_{\vec{u}_0} \setminus \texttt{initialValue}_{\vec{u}_0}$ (that is no method of $P_{\vec{u}_0}$ except for $\texttt{initialValue}_{\vec{u}_0}$) has been changed, so the control flow is unchanged and thus no control dependencies have changed. So $n_1 \longrightarrow n_2$ cannot be a control dependency.
– *data dependence edge*: Data dependencies are either caused through use and definition of variables or

references and modification to heap locations. $P_{\vec{u}} \setminus$ `initialValue`$_{\vec{u}}$ contains no new statements and thus no additional uses and definitions of variables. So the additional data dependence cannot stem from them and has to be introduced through accesses to heap locations. The only heap locations that have changed are the elements of `list`. Our restrictions forbid references to this list in P_{main}, so $n_1 \rightarrow n_2$ cannot be a data dependency.

- *call edge*: $P_{\vec{u}} \setminus$ `initialValue`$_{\vec{u}}$ contains no new statements and thus no new call statements. So $n_1 \rightarrow n_2$ cannot be a call dependency.
- *parameter and heap dependence edge*: These edges are caused by passing parameters into methods and returning method results. As $P_{\vec{u}} \setminus$ `initialValue`$_{\vec{u}}$ is not changed, no method signatures and calls are altered. So $n_1 \rightarrow n_2$ cannot be caused by additionally passed parameter or returned results. The last remaining possibility is that the edge is caused by a change in the referenced or modified heap locations. Due to the restrictions we know that only heap locations referring to elements of `list` may have changed. As P_{main} does not contain references to `list` or its elements, $n_1 \rightarrow n_2$ is no parameter or heap dependency.

We showed that $n_1 \rightarrow n_2$ cannot exists, thus path p cannot exist and so no illegal information flow can be introduced by changes in `initialValue`$_{\vec{u}_0}$.