Extending and Applying a Framework for the Cryptographic Verification of Java Programs

Ralf Küsters[1], Enrico Scapin[1], Tomasz Truderung[1], and Jürgen Graf[2]

[1] University of Trier, Germany
[2] Karlsruhe Institute of Technology, Germany
{kuesters,scapin,truderung}@uni-trier.de, graf@kit.edu

Abstract. In our previous work, we have proposed a framework which allows tools that can check standard noninterference properties but a priori cannot deal with cryptography to establish cryptographic indistinguishability properties, such as privacy properties, for Java programs. We refer to this framework as the CVJ framework (Cryptographic Verification of Java Programs) in this paper.

While so far the CVJ framework directly supports public-key encryption (without corruption and without a public-key infrastructure) only, in this work we further instantiate the framework to support, among others, public-key encryption and digital signatures, both with corruption and a public-key infrastructure, as well as (private) symmetric encryption. Since these cryptographic primitives are very common in security-critical applications, our extensions make the framework much more widely applicable.

To illustrate the usefulness and applicability of the extensions proposed in this paper, we apply the framework along with the tool Joana, which allows for the fully automatic verification of noninterference properties of Java programs, to establish cryptographic privacy properties of a (non-trivial) cloud storage application, where clients can store private information on a remote server.

1 Introduction

In [24], a framework has been proposed which allows tools that can check standard noninterference properties but cannot deal with cryptography directly, in particular probabilities and polynomially bounded adversaries, to establish cryptographic indistinguishability properties, such as privacy properties, for Java programs. In this paper, we refer to this framework as the CVJ framework (Cryptographic Verification of Java programs). The framework combines techniques from program analysis and cryptography, more specifically, universal composability [9,19,27,29], a well-established concept in cryptography. The idea is to first check noninterference properties for the Java program to be analyzed where cryptographic operations (such as encryption) are performed within so-called ideal functionalities. Such functionalities typically provide guarantees even in the face of unbounded adversaries and can often be formulated without probabilistic operations. Therefore, such analysis can be carried out by tools that a priori cannot deal with cryptography (probabilities, polynomially bounded adversaries). Theorems shown within the framework now imply that the Java program enjoys strong cryptographic indistinguishability properties when the ideal functionalities are replaced by their realizations, i.e., the actual cryptographic operations.

M. Abadi and S. Kremer (Eds.): POST 2014, LNCS 8414, pp. 220–239, 2014.

The theorems proved within the CVJ framework are very general in that they guarantee that any ideal functionality can be replaced by its realization. In particular, they are not tailored to specific cryptographic operations. However, to make the framework directly applicable to a wide range of cryptographic software, i.e., software that uses cryptographic operations (such as asymmetric and symmetric encryption, digital signatures, MACs, etc.), it is necessary to provide a rich set of ideal functionalities along with their realizations written in Java. So far, in [24] only an ideal functionality for public-key encryption has been proposed and it has been shown that this functionality can be realized by any IND-CCA2-secure public-key encryption scheme, a standard security notion for such schemes (see, e.g., [4]). This functionality does not support reasoning about corruption and also it does not support a public-key infrastructure (PKI).

Contribution of This Paper. The main goal and the main contribution of this work is therefore to instantiate the CVJ framework with further (and more suitable) ideal functionalities which commonly occur in cryptographic applications, and to provide realizations of such functionalities based on standard cryptographic assumptions. We note that similar functionalities as the once introduced in this work have been considered in the cryptographic literature based on Turing machine models (see, e.g., [9, 26, 29]) before. The new contribution here is that we provide formulations in *Java* (more precisely, in a rich fragment of Java) such that these functionalities can actually be used to analyze Java programs. Designing such functionalities and carrying out the proofs (w.r.t. programming language semantics) is non-trivial and requires some care since the interaction between different classes is much more complex than between Turing machines, where in the former case we have to deal, for example, with exceptions, inheritance, references to potential complex objects that can be exchanged, and hence, the manipulation of one object can affect many other objects. Also, since the ideal functionalities we propose will be part of the (Java) programs to be analyzed, they should be formulated in a "tool friendly" way. For example, for this reason, in our functionalities corruption is modeled in a quite different way than it is typically done in the Turing machine models.

More concretely, in this work we propose ideal functionalities, written in Java, for public-key encryption, digital signatures, (private) symmetric encryption, and nonce generation.

The functionalities for public-key encryption and digital signatures support static corruption and a public-key infrastructure. The latter means that parties can register their public encryption and verification keys using the functionalities. Other parties can then use the functionalities to encrypt messages and verify signatures by simply providing the name of the intended recipient of the message/the alleged signer of the message. The functionality then guarantees that the correct public-key is used for encryption/verification. As for static corruption, the adversary can register his own (possibly dishonestly generated) public keys which then can be used by other (honest) parties just like honestly generated and registered keys. We show that both functionalities, public-key encryption and digital signatures, can be realized using standard cryptographic schemes and assumptions (IND-CCA2-secure public-key encryption schemes and UF-CMA-secure digital signature schemes).

The functionality for private symmetric encryption allows a user to encrypt messages (using a symmetric encryption scheme) for herself. She does not share the symmetric key with other parties. This is useful, for example, to store confidential information on an untrusted medium. Again, this functionality is realized using a standard symmetric encryption scheme, based on standard cryptographic assumptions (IND-CCA2 security).

Finally, the ideal functionality for nonce generation that we propose guarantees that nonces are always fresh. That is, this functionality prevents collisions of nonces. It is realized in the obvious way, by choosing nonces (of the length of the security parameter) uniformly at random.

We illustrate the usefulness and applicability of these functionalities in a case study. We apply the CVJ framework, along with the tool Joana [16, 17], which allows for the fully automatic verification of noninterference properties of Java programs, to establish cryptographic privacy properties of a non-trivial cloud storage application, where clients can store private information on a remote server. The cloud storage system makes use of all cryptographic primitives considered in this paper, and hence, the code of these functionalities is included in the verified program. We note that, except for a much simpler Java program analyzed in [24], there has been no other verification effort that establishes cryptographic security guarantees of Java programs.

Related Work. Obtaining cryptographic guarantees for programs written in real-world programming languages is a challenging and quite recent research field (see also [24] for a discussion of related work). Many approaches in this field carry out symbolic (Dolev-Yao style) analysis, without computational/cryptographic guarantees (see, e.g., [5, 11, 15]). Most, of the very few, approaches that aim at cryptographic guarantees follow one of the following approaches: i) They rely on symbolic analysis and then apply computational soundness results (see, e.g., [1]), ii) they derive formal models from the source code and analyze these models using specialized tools for cryptographic verification, such as the tool CryptoVerif [7] (see, e.g., [2]), or iii) they derive source code from formal specifications (see, e.g., [8]). The CVJ framework, in contrast, aims at using existing program analysis tools and techniques to directly obtain cryptographic security guarantees. It is the only approach for the cryptographic analysis of Java programs, other approaches aim at C or F# code. Also, unlike most other approaches, it considers cryptographic indistinguishability properties, rather than trace properties, such as authentication and weak secrecy. An approach similar to the approach taken in the CVJ framework is the one by Fournet et al. [6, 12]. However, they consider F# and focus on the use of refinement types.

Structure of This Paper. In Section 2, we first briefly recall the CVJ framework. In the four subsequent sections, we present the ideal functionalities for public-key encryption, digital signatures, private symmetric encryption, and nonce generation, respectively, including their realizations. In Section 7, we turn to the case study. Further details are provided in the extended version of this paper [22].

2 The CVJ Framework

We briefly recall the framework from [24]. The definitions and theorems stated here are somewhat simplified and informal, but should suffice to follow the rest of the paper. We refer the reader to [24] for full details.

As already mentioned in the introduction, in order to establish cryptographic indistinguishability properties for a Java program, by the CVJ framework it suffices to prove that the program enjoys a (standard) noninterference property when the cryptographic operations are replaced by so-called ideal functionalities, which in our case will model cryptographic primitives, such as encryption and digital signatures. The CVJ framework then ensures that the Java program enjoys the desired cryptographic indistinguishability properties when the ideal functionalities are replaced by their realizations, i.e., the actual cryptographic operations. Since ideal functionalities often do not involve probabilistic operations and are secure even for unbounded adversaries, the noninterference properties can be verified by tools that a priori cannot deal with cryptography (probabilities, polynomially bounded adversaries). Without the ideal functionalities, the tools would, for example, consider a secret message that is sent encrypted over a network controlled by the adversary to be an information leakage, because an unbounded adversary can break the encryption.

Jinja+. The CVJ framework is stated and proven for a Java-like language called *Jinja+*. Jinja+ is based on *Jinja* [18] and extends this language with some useful additional features, such as arrays and randomness. Jinja+ covers a rich subset of Java, including classes, inheritance, (static and non-static) fields and methods, the primitive types `int`, `boolean`, and `byte` (with the usual operators for these types), arrays, exceptions, and field/method access modifiers, such as `public`, `private`, and `protected`. It also includes the primitive `randomBit()` which returns a random bit each time it is called.

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

All Java programs considered in this paper, including the systems considered in our case study as well as the functionalities fall into the Jinja+ fragment. While the syntax of Jinja+ and Java differ, their is a straightforward translation from Jinja+ to Java, which is why we use Java syntax throughout this paper.

Indistinguishability. An *interface I* is defined like a (Jinja+) system but where (i) all private fields and private methods are dropped and (ii) method bodies as well as static field initializers are dropped. A system S *implements* an interface I, written $S : I$, if I is a subinterface of the public interface of S, i.e. the interface obtained from S by dropping method bodies, initializers of static fields, private fields, and private methods. We say that *a system S uses an interface I*, written $I \vdash S$, if, besides its own classes, S uses at most classes/methods/fields declared in I. We write $I_0 \vdash S : I_1$ for $I_0 \vdash S$ and $S : I_1$. We also say that two interfaces are *disjoint* if the sets of class names declared in these interfaces are disjoint.

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

A system E is called an *environment* if it declares a distinct private static variable `result` of type `boolean` with initial value `false`. Given a system $S : I$, we call E an

I-environment for S if there exists an interface I_E disjoint from I such that $I_E \vdash S : I$ and $I \vdash E : I_E$. Note that $E \cdot S$ is a complete program. The value of the variable `result` at the end of the run of $E \cdot S$ is called the *output* of the program $E \cdot S$; the output is `false` for infinite runs. If $E \cdot S$ is a deterministic program, we write $E \cdot S \rightsquigarrow \texttt{true}$ 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`.

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

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

Let P_1 and P_2 be (complete, possibly probabilistic) programs. We say that P_1 and P_2 are *computationally equivalent*, written $P_1 \equiv_{\mathsf{comp}} P_2$, if $|\mathsf{Prob}\{P_1(\eta) \rightsquigarrow \texttt{true}\} - \mathsf{Prob}\{P_2(\eta) \rightsquigarrow \texttt{true}\}|$ is a negligible function in the security parameter η.[1]

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

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

More formally, let F and R be probabilistic polynomially bounded systems which implement the same interface I_{out} and use the same interface I_E, except that in addition F may use some interface I_S provided by a simulator. Then, we say that R *realizes F w.r.t. I_{out}*, written $R \leq^{I_{out}} F$ or simply $R \leq F$, if there exists a probabilistic polynomially bounded system S (the simulator) such that $R \approx^{I_{out}}_{\mathsf{comp}} S \cdot F$. As shown in [24], $\leq$ is reflexive and transitive.

A main advantage of defining security of real systems by the realization relation $\leq$ is that systems can be analyzed and designed in a modular way: The following theorem implies that it suffices to prove security for the systems R_0 and R_1 separately in order to obtain security of the composed system $R_0 \cdot R_1$.

Theorem 1 (Composition Theorem (simplified) [24]). *Let I_0 and I_1 be disjoint interfaces and let R_0, F_0, R_1, and F_1 be probabilistic polynomially bounded systems such*

[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 $\eta > \eta_0$.

that $R_0 \leq^{I_0} F_0$ and $R_1 \leq^{I_1} F_1$. Then, $R_0 \cdot R_1 \leq^{I_0 \cup I_1} F_0 \cdot F_1$, where $I_0 \cup I_1$ is the union of the class, method and field names declared in I_0 and I_1.

Noninterference. The (standard) noninterference notion for confidentiality [13] requires the absence of information flow from high to low variables within a program. Here, we define noninterference for a deterministic (Jinja+) program P with some static variables $\vec{x}$ 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 variables* $\vec{x}$ (and low variables). By $P[\vec{a}]$ we denote the program P where the high variables $\vec{x}$ are initialized with values $\vec{a}$ and the low variables are initialized as specified in P.

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

The above notion of noninterference deals with complete programs (closed systems). This notion is generalized to open systems as follows: Let I be an interface and let $S[\vec{x}]$ be a (not necessarily closed) deterministic system with a security parameter and high variables $\vec{x}$ 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 the only low variable. Note that here neither E nor S are required to be polynomially bounded.

Tools for checking noninterference often consider only a single closed program. However, I-noninterference is a property of a potentially open system $S[\vec{x}]$, which is composed with an arbitrary I-environment. Therefore, in [24] a technique has been developed which reduces the problem of checking I-noninterferent to checking noninterference for a single (almost) closed system. More specifically, it was shown that to prove I-noninterference for a system $S[\vec{x}]$ with $I_E \vdash S : I$ it suffices to consider a single environment $\tilde{E}^{I,I_E}_{\vec{u}}$ (or $\tilde{E}_{\vec{u}}$, for short) only, which is parameterized by a sequence $\vec{u}$ of values. The output produced by $\tilde{E}_{\vec{u}}$ to $S[\vec{x}]$ is determined by $\vec{u}$ and is independent of the input it gets from $S[\vec{x}]$. To keep $\tilde{E}_{\vec{u}}$ simple, the analysis technique assumes some restrictions on interfaces between $S[\vec{x}]$ and E. In particular, $S[\vec{x}]$ and E should interact only through primitive types, arrays, exceptions, and simple objects. Moreover, E is not allowed to call methods of S directly (formally, we require I to be $\emptyset$). However, since S can call methods of E, this is not an essential limitation.

Theorem 2 (simplified, [24]). *Let $S[\vec{x}]$ be a deterministic program with a restricted interface to its environment, as mentioned above, and let $I = \emptyset$. Then, I-noninterference holds for $S[\vec{x}]$ if and only if for all sequences $\vec{u}$ noninterference holds for $\tilde{E}_{\vec{u}} \cdot S[\vec{x}]$.*

Automatic analysis tools, such as Joana [16, 17], often ignore or can ignore specific values encoded in a program, such as an input sequence $\vec{u}$. Hence, such an analysis of $E_{\vec{u}} \cdot S[\vec{x}]$ implies noninterference for all sequences $\vec{u}$, and by the theorem, this implies I-noninterference for $S[\vec{x}]$.

From I-Noninterference to Computational Indistinguishability. The central theorem that immediately follows from (the more general) results proven within the CVJ framework is the following.

Theorem 3 (simplified, [24]). *Let I and J be disjoint interfaces. Let F, R, $P[\vec{x}]$ be systems such that $R \leq^J F$, $P[\vec{x}] \cdot F$ is deterministic, and $P[\vec{x}] \cdot F : I$ (and hence, $P[\vec{x}] \cdot R : I$). Now, if $P[\vec{x}] \cdot F$ is I-noninterferent, then, for all $\vec{a}_1$ and $\vec{a}_2$ (of appropriate type), we have that $P[\vec{a}_1] \cdot R \approx^I_{\mathsf{comp}} P[\vec{a}_2] \cdot R$.*

The intuition and the typical use of this theorem is that the cryptographic operations that P needs to perform are carried out using the system R (e.g., a cryptographic library). The theorem now says that to prove cryptographic privacy of the secret inputs ($\forall\ \vec{a}_1, \vec{a}_2$: $P[\vec{a}_1] \cdot R \approx^J_{\mathsf{comp}} P[\vec{a}_2] \cdot R$) it suffices to prove I-noninterference for $P[\vec{x}] \cdot F$, i.e., the system where R is replaced by the ideal counterpart F (the ideal cryptographic library). The ideal functionality F, which in our case will model cryptographic primitives in an ideal way, can typically be formulated without probabilistic operations and also the ideal primitives specified by F will be secure even in presence of unbounded adversaries. Therefore, the system $P[\vec{x}] \cdot F$ can be analyzed by standard tools that a priori cannot deal with cryptography (probabilities and polynomially bounded adversaries).

As mentioned before, F relies on the interface $I_E \cup I_S$ (which, for example, might include an interface to a network library) provided by the environment and the simulator, respectively. This means that when checking noninterference for the system $P[\vec{x}] \cdot F$ the code implementing this library does not have to be analyzed. Being provided by the environment/simulator, it is considered completely untrusted and the security of $P[\vec{x}] \cdot F$ does not depend on it. In other words, $P[\vec{x}] \cdot F$ provides noninterference for all implementations of the interface. Similarly, R relies on the interface I_E provided by the environment. Hence, $P[\vec{x}] \cdot R$ enjoys computational indistinguishability for all implementations of I_E. This has two advantages: i) one obtains very strong security guarantees and ii) the code to be analyzed in order to establish noninterference/computational indistinguishability is kept small, considering the fact that libraries tend to be very big.

3 Public-Key Encryption with a Public Key Infrastructure

We now propose an ideal functionality Ideal-PKIEnc, formulated in Java (Jinja+), for public-key encryption with a public-key infrastructure (PKI). This functionality is an extension of a more restricted public-key encryption functionality proposed in [24]. First, the functionality proposed here allows a user to encrypt messages for a given party based on the identifier of this party. The functionality uses the included public key infrastructure to obtain the public key of the party registered under the given identifier. In contrast, to encrypt a message, the user of the functionality in [24] had to provide a public-key herself, and hence, take care of the correct binding of public keys to parties herself. Second, in the functionality proposed here, as opposed to the one in [24], we model static corruption, including dishonestly generated keys. For this, special care was needed to make sure that the resulting functionality is "tool-friendly".

We also provide an implementation (realization) of this ideal functionality, denoted by Real-PKIEnc, in Java (Jinja+) and prove, within the CVJ framework, that this implementation realizes the ideal functionality Ideal-PKIEnc under standard cryptographic assumptions.

As already mentioned in the introduction, the design of such functionalities and the realization proofs pose additional challenges compared to the Turing machine based formulations proposed in the cryptographic literature.

In the rest of this section, we first provide the interface for Ideal-PKIEnc, and hence, Real-PKIEnc. Then, the actual ideal functionality and its realization are presented, along with a realization theorem.

3.1 The Interface for Public-Key Encryption

In this section, we present the interface I_{PKIEnc} of the ideal functionality Ideal-PKIEnc and its implementation Real-PKIEnc and discuss the intended way of using it. The interface I_{PKIEnc} is specified as follows:

```
 1  public class Encryptor {
 2    public Encryptor(byte[] publicKey);
 3    public byte[] encrypt(byte[] message);
 4    public byte[] getPublicKey();
 5  }
 6  public final class Decryptor {
 7    public Decryptor();
 8    public byte[] decrypt(byte[] message);
 9    public Encryptor getEncryptor();
10  }
11  public class RegisterEnc {
12    public static void registerEncryptor(int id, Encryptor encryptor,
13                        byte[] pki_domain) throws PKIError, NetworkError;
14    public static Encryptor getEncryptor(int id, byte[] pki_domain)
15                                              throws PKIError, NetworkError;
16  }
```

Typical Usage. The intended way for an honest user with identifier ID_A to create and register her keys is the following:

```
17  Decryptor decryptor = new Decryptor();
18  Encryptor encryptor = decryptor.getEncryptor();
19  try {
20      RegisterEnc.registerEncryptor(ID_A, encryptor, PKI_DOMAIN);
21  }
22  catch (PKIError e) {}      // registration failed: id already claimed
23  catch (NetworkError e) {}  // network problems
```

Intuitively, an object of class Decryptor encapsulates a public/private key pair, generated when the object is created (line 17 above). This object provides access to the method decrypt. The owner of this object (that is, the party who has created it) is not supposed to share it with any other parties. Instead, the owner of the decryptor shares an associated encryptor (obtained in line 18), which, intuitively, encapsulates only the public key. More precisely, to make her public key available within a PKI to other parties, the user registers the encryptor she has obtained (line 20). That is, she registers her encryptor under her identifier (ID_A) and what we call a PKI domain (which is a publicly known identifier used to distinguish keys registered for different purposes/applications). This step may result in an error: i) if some key has been registered already under this identifier and PKI domain (exception PKIError), or ii) if some network failure occurred, e.g., the registration server was unavailable (exception NetworkError). We emphasize that we do not require the party who wants to register a public key to provide a proof of possession (PoP) of the private key corresponding to the public key.[2] After an encryptor has been registered, it can be used by other parties as follows:

```
24 try {
25   Encryptor encryptor = RegisterEnc.getEncryptor(ID_A, PKI_DOMAIN);
26   encryptor.encrypt(message);
27 } catch(PKIError e) {}     // id has not been successfully registered
28   catch(NetworkError e) {} // network problems
```

The encryptor of the party registered under `ID_A` and `PKI_DOMAIN` is obtained in line 25 and used in line 26 to encrypt a message. Note that a user can also obtain the public key encapsulated in the encryptor, using the method `getPublicKey`.

Corruption. To model (static) corruption, we allow encryptors also to be created directly, without creating associated decryptors, simply by providing an arbitrary bitstring `pubk` as the public key:

```
29 Encryptor enc = new Encryptor(pubk);
30 try {
31     RegisterEnc.registerEncryptor(ID, enc, PKI_DOMAIN);
32 } catch (PKIError | NetworkError e) {}
```

By this, a dishonest party (the adversary) can register any bitstring `pubk` as a public key, including dishonestly generated keys. This key can then be used by any other party (honest and dishonest) to encrypt messages for the dishonest party, just like public keys of honest parties. Note that since we do not require PoPs, a dishonest party can register any public key of another (possibly honest) party under his identity. (As mentioned before, the literature on PKIs recommends that applications should not rely on PoPs being performed [3].)

An encryptor created in the above way is called *corrupted*. There is no corresponding (corrupted) decryptor, because the adversary can run the decryption algorithm himself. For messages encrypted with a corrupted encryptor (public key), no security guarantees are provided. (Jumping ahead to Section 3.2, the functionality will hand the message to be encrypted with a corrupted encryptor directly to the environment/adversary/simulator.)

We note that, as expected, when some party obtains an encryptor by the method `RegisterEnc.getEncryptor`, the party does not know a priori whether the obtained encryptor is corrupted (it has been generated directly) or uncorrupted (it has been generated via `Decryptor`).

3.2 The Ideal Functionality for Public-Key Encryption

We now present the ideal functionality for public-key encryption, Ideal-PKIEnc. This functionality provides the interface I_{PKIEnc}, introduced above, to its users (parties, environment) with ideal implementations of the methods declared in I_{PKIEnc}.

[2] In most applications, PoPs are not necessary and as argued in the literature (see, e.g., [3]), applications should be designed in such a way that their security does not depend on the assumption of such proofs being performed.

The functionality Ideal-PKIEnc is defined on top of the interface $I_{\mathsf{CryptoLibEnc}}$ which contains methods for key generation, encryption, and decryption:

```
33 public class CryptoLib {
34   public static KeyPair pke_generateKeyPair();
35   public static byte[] pke_encrypt(byte[] message, byte[] publicKey);
36   public static byte[] pke_decrypt(byte[] ciphertext, byte[] privKey);
37 }
```

So Ideal-PKIEnc expects the above methods to be implemented outside of Ideal-PKIEnc. In the analysis of a system $P[\vec{x}]$ which uses Ideal-PKIEnc (i.e., in the analysis of the system $P[\vec{x}] \cdot$ Ideal-PKIEnc), such methods have to be provided by the environment, and thus, are completely untrusted. In particular, in the analysis of $P[\vec{x}] \cdot$ Ideal-PKIEnc the code for CryptoLib, which would typically be very large, does not have to be analyzed. This tremendously simplifies the analysis of $P[\vec{x}] \cdot$ Ideal-PKIEnc (see also the explanation in Section 2 following Theorem 3).

The basic idea of the implementation of Ideal-PKIEnc is that if a message m is to be encrypted with an (uncorrupted) public key, then not m but a sequence of zeros of the same length as m is encrypted instead, using method pke_encrypt of CryptoLib. By this, it is guaranteed that the resulting ciphertext c does not depend on m, except for the length of m. The functionality stores the pair (m,c) for later decryption. If some ciphertext c' is to be decrypted, the functionality first checks whether there exists a pair of the form (m',c') (the functionality guarantees that there is at most one such pair). Then, m' is returned as the plaintext. If no such pair exists (and hence, c' was not created using the functionality), c' is decrypted using method pke_decrypt of CryptoLib, and the resulting plaintext is returned. More specifically, Ideal-PKIEnc works as follows.

On initialization of an object of the class Decryptor, a public/private key pair is created by calling the key generation method of the class CryptoLib. At this point, the decryptor object also creates an (initially empty) list of message/ciphertext pairs. This list is used as a look-up table for decryption by the method decrypt of class Decryptor as sketched above.

Encryptors returned by the method getEncryptor of class Decryptor are objects of the class UncorruptedEncryptor (which is a subclass of the class Encryptor). An encryptor object contains the same public-key as the associated decryptor and shares (a reference to) the list of message/ciphertext pairs with the associated decryptor. When method encrypt of such an encryptor is called with a message m, the encryption method of class CryptoLib is called to encrypt a sequence of zeros of the same length as m, resulting in a ciphertext c (ciphertexts seen before are rejected). Then, the pair (m,c) is stored in the list and the ciphertext c is returned as the result of the encryption.

In contrast, a corrupted encryptor (i.e., an encryptor object created directly as in line 29 above, rather than being derived from a decryptor) implements encryptions simply by calling the encryption method of the class CryptoLib using the bitstring (the public key) it has been provided with upon creation. Note that in this case, no security guarantees are provided; the original message instead of zeros is encrypted.

The methods for registering and obtaining encryptors in class RegisterEnc are implemented in a straightforward way by Ideal-PKIEnc, using a list of registered encryptors along with associated identifiers and domains.

The most important part of the code of Ideal-PKIEnc is listed in the extended version of this paper [22]; for the full code see [23].

3.3 The Realization of Ideal-PKIEnc

We now provide the realization Real-PKIEnc of the ideal functionality Ideal-PKIEnc presented above.

The functionality Real-PKIEnc builds on a public key infrastructure. A public-key infrastructure is a trusted public key registry, where i) users can register their public keys under their identifiers and (PKI) domains (in the sense of Section 3.1) and ii) users can obtain other users' public keys by providing the identifiers and domains of these users. The interface I_{PKI} for the public key infrastructure used by Real-PKIEnc is the following:

```
public class PKI {
  static void register(int id, byte[] domain, byte[] pubKey)
                                             throws PKIError, NetworkError;
  static byte[] getKey(int id, byte[] domain)
                                             throws PKIError, NetworkError;
}
```

The method `register` is supposed to throw `PKIError` if the provided user identifier and domain pair has been claimed already, i.e., some other party has registered a key for the same identifier and domain pair before. The same exception is supposed to be thrown by the method `getKey` if the given identifier `id` has not been registered. Registering or fetching a public key typically involves to contact a public-key server. If this fails, the `NetworkError` is thrown. When proving that Real-PKIEnc realizes Ideal-PKIEnc we will assume that I_{PKI} is properly implemented (see Section 3.4 for details).

Now, based on I_{PKI}, the different classes and methods provided by Real-PKIEnc are implemented as presented next.

The methods `registerEncryptor` and `getEncryptor` of the class `RegisterEnc` work as follows. When an encryptor is to be registered by the method `registerEncryptor`, its public key is registered in the PKI using the method `register`. The method `getEncryptor` uses the method `getKey` to fetch the corresponding public key and wraps it into an encryptor which is then returned.

The classes `Encryptor` and `Decryptor` of Real-PKIEnc are implemented in a straightforward way using an encryption scheme: messages are simply encrypted/decrypted directly using such a scheme. Note that whether an encryptor was obtained from a decryptor (using the method `getEncryptor`) or whether it was created directly (as in line 29) leads to the same implementation, namely, invoking the encryption function of the encryption scheme. The only difference is that in one case the public/private key pair was created (honestly) within the class `Decryptor` of Real-PKIEnc and in the other case the public key was created outside of Real-PKIEnc (possibly in some dishonest way).

The most important part of the code of Real-PKIEnc is listed in [22]; see [23] for the full code.

3.4 Realization Result

We now show that Real-PKIEnc realizes Ideal-PKIEnc, provided that i) the encryption scheme used in the implementation of Real-PKIEnc is IND-CCA2-secure [4] and ii) that the public-key infrastructure used by Real-PKIEnc works "properly".

As for i), we note that IND-CCA2-security is a standard and widely used security notion for public-key encryption schemes. Similarly to ideal functionality for public-key encryption proposed in the cryptographic literature, it has been shown that IND-CCA2-security is necessary to realize Ideal-PKIEnc (see, e.g., [9,26]).

As for ii), the behavior of a "proper public-key infrastructure" is formalized by an ideal functionality Ideal-PKI, which operates in the obvious way: It maintains a list of registration records, each consisting of an identifier, a domain, and a key (the code is given in [22]). The adversary (simulator) is informed about registration requests and requests for obtaining public-keys and can schedule when these requests are answered by Ideal-PKI (because in a realization such requests typically involve communication over a network controlled by the adversary). We assume the existence of some public-key infrastructure Real-PKI that realizes Ideal-PKI. Note that there are various ways of realizing Ideal-PKI and that all of them will require certain trust assumptions. For example, one could assume the existence of one or more honest certificate authorities and that parties are provided with the (authentic) public keys of these authorities. Typically, one would use some existing public-key infrastructure (with appropriate assumptions) to realize Ideal-PKI. However, this is not the focus of this work. (In fact, proving the security of a full-fledged PKI would be a challenging task by itself.). In our case study (see Section 7), we consider a simple realization which involves a single certificate authority, the assumption being that it in fact realizes Ideal-PKI.

With this, we can now state our main theorem for public-key encryption.

Theorem 4. *If* Real-PKIEnc *uses an IND-CCA2-secure public-key encryption scheme and* Real-PKI $\leq^{I_{PKI}}$ Ideal-PKI*, then* Real-PKIEnc $\cdot$ Real-PKI $\leq^{I_{PKIEnc}}$ Ideal-PKIEnc.

The proof of Theorem 4 is given in [22]. The proof is highly modular and leverages such properties of the realization relation as the composition theorem, reflexivity, and transitivity. In the proof, we split Ideal-PKIEnc and Real-PKIEnc into two parts: one providing encryption and decryption and one providing key registration and retrieving. For the former part, we generalize the result of [24] for public-key functionality without corruption and without PKI to the case with corruption.

4 Digital Signatures with a Public Key Infrastructure

In this section, we propose an ideal functionality Ideal-Sig, formulated in Java (Jinja+), for digital signatures with a public key infrastructure, where, again, we model corruption. We also provide a real implementation Real-Sig of this functionality in Java (Jinja+) and prove, in the CVJ framework, that it realizes Ideal-Sig. Just as for public key encryption, similar functionalities for digital signatures have been proposed in the cryptographic literature before (see, e.g., [10,26]). But again, the new contribution here is that we provide a formulation in Java, instead of the (simpler) Turing machine models, such that these functionalities can actually be used to analyze Java programs. This is non-trivial and needs some care. We first present the public interface of Ideal-Sig and Real-Sig.

4.1 The Interface for Digital Signatures

The public interface I_{PKISig} of Ideal-Sig and Real-Sig (both have the same public interface) is as follows:

```
 1 public final class Signer {
 2   public Signer();
 3   public byte[] sign(byte[] message);
 4   public Verifier getVerifier();
 5 }
 6 public class Verifier {
 7   public Verifier(byte[] verifKey);
 8   public boolean verify(byte[] signature, byte[] message);
 9   public byte[] getVerifKey();
10 }
11 public class RegisterSig {
12   public static void registerVerifier(int id, Verifier verifier,
13                         byte[] pki_domain) throws PKIError, NetworkError;
14   public static Verifier getVerifier(int id, byte[] pki_domain)
15                                          throws PKIError, NetworkError;
16 }
```

Typical Usage. Similarly to public-key encryption, the intended way for an honest user with identifier ID_A to create and register her keys is the following:

```
17 Signer sig = new Signer();
18 Verifier ver = sig.getVerifier();
19 try {
20     SigEnc.registerVerifier(ID_A, ver, PKI_DOMAIN);
21 } catch (PKIError e) {}      // registration failed: id already claimed
22   catch (NetworkError e) {}  // network problems
```

Intuitively, an object of the class Signer encapsulates a verification/signing key pair, which is generated when the object is created (line 17). It allows a party who owns such an object to sign messages (this requires the signing key), using the method sign (of the class Signer). This party can also obtain a Verifier object (line 18), which encapsulates the related verification key and can be used (by other parties) to verify signatures via the method verify. Similarly to the case of public-key encryption, such a verifier can be registered in the public-key infrastructure (line 20) in order to make the verification key available to other parties. Again, we do not require a proof of possession of the corresponding signing key.

After a verifier has been registered, it can be used by other parties to check whether a signature signature is valid for a message message w.r.t. the verification key of (ID_A, PKI_DOMAIN) encapsulated in verifier:

```
try {
  Verifier verifier = RegisterSig.getVerifier(ID_A, PKI_DOMAIN);
  verifier.verify(signature, message);
} catch(PKIError e) {}     // id has not been successfully registered
  catch(NetworkError e) {} // network problems
```

Corruption. To model (static) corruption, analogously to the case of public-key encyrption we allow verifiers to be created directly, without creating associated signers, simply by providing an arbitrary bitstring `verif_key` as the public key:

```
Verifier ver = new Verifier(verif_key);
try {
    RegisterSig.registerVerifier(ID, ver, PKI_DOMAIN);
} catch (PKIError | NetworkError e) {}
```

By this, a dishonest party (the adversary) can register any bitstring `verif_key` he wants as a verification key, including dishonestly generated keys. This key can then be used by any other party (honest and dishonest) to verify messages signed by the dishonest party, just like with verification keys of honest parties. Note that since we do not require PoPs, a dishonest party can register any verification key of another (possibly honest) party under his identity. A verifier created in such a way is called *corrupted*. A corresponding signing object is not necessary as the adversary can directly sign messages by himself using the matching signing key (if this key is known to the adversary). Note that, given a verifier object, other parties cannot tell a priori whether this verifier object is corrupted or not.

4.2 The Ideal Functionality for Digital Signatures

We now present the ideal functionality for digital signatures, Ideal-Sig. This functionality provides the interface I_{PKISig}, introduced above, to its users (parties, environment) with ideal implementations of the methods declared in I_{PKISig}.

The functionality is defined on top of the interface $I_{\mathsf{CryptoLibSig}}$ which contains methods for key generation, signing, and verification. Analogously to the interface $I_{\mathsf{CryptoLibEnc}}$ for public-key encryption, these methods are supposed to be provided by the environment, and hence, are completely untrusted. In particular, in the analysis of a system that uses Ideal-Sig, they do not have to be analyzed, which, again, greatly simplifies the analysis task.

Now, Ideal-Sig works as follows. On initialization of an object of class `Signer`, a verification/signing key pair is created by calling the key generation operation of the interface $I_{\mathsf{CryptoLibSig}}$. A signer object also creates an (initially empty) list of signed messages; this list will be shared with all associated verifiers (objects returned by `getVerifier`). When the method `sign` is called to sign a message m, the signing procedure of $I_{\mathsf{CryptoLibSig}}$ is called to sign m using the encapsulated signing key. Before this signature is returned, the signed message m is added to the list of signed messages.

A verifier object returned by the method `getVerifier` belongs to the class `UncorruptedVerifier` (a subclass of the class `Verifier`) and it implements ideal verification as follows: the method `verify` when called to verify a signature s on a message m first uses the verification procedure of $I_{\mathsf{CryptoLibSig}}$ to check if s is a valid signature on m w.r.t. the verification key encapsulated in the verifier object. If this is the case, it additionally checks if *m* is in the list of signed messages (this list, as mentioned before, is shared with the associated signer object). If this is true as well, the method returns 'true'. The idea behind this procedure is that, independently of how the signing and verification algorithms work, the verification of a signature on some message succeeds only if this message has been signed before (and hence, logged) using Ideal-Sig.

A (corrupted) verifier object created directly implements the verification procedure simply by calling the verification method of $I_{\mathsf{CryptoLibSig}}$.

The methods for registering and obtaining verifiers in class `RegisterSig` are implemented in a straightforward way by Ideal-PKIEnc, using a list of registered verifiers along with associated identifiers and domains.

The most important part of the code of Ideal-Sig is listed in [22]; see [23] for the full code.

4.3 The Realization of Ideal-Sig

The classes `Verifier` and `Signer` of the realization Real-Sig of the ideal functionality Ideal-Sig are implemented in a straightforward way using a digital signature scheme: messages are simply signed/verified directly using such a scheme. Analogously to the methods in `EncPKI`, the methods `registerVerifier` and `getVerifier` of the class `RegisterSig` are based on the interface I_{PKI} introduced in Section 3.3.

The most important part of the code of Real-Sig is listed in [22]; see [23] for the full code.

4.4 Realization Result

We prove that Real-PKISig realizes Ideal-PKISig, provided that i) the signature scheme used in the implementation of Real-PKISig is UF-CMA-secure [14] and ii) that, analogously to the case of public-key encryption, the public-key infrastructure used by Real-PKISig realizes the ideal functionality Ideal-PKI (see Section 3.4). Again, it has been shown that UF-CMA-security is necessary to realize Ideal-PKIEnc (see, e.g., [26]).

Theorem 5. *If* Real-PKISig *uses an UF-CMA-secure signature scheme and* $\mathsf{Real\text{-}PKI} \leq^{I_{\mathsf{PKI}}} \mathsf{Ideal\text{-}PKI}$*, then* $\mathsf{Real\text{-}PKISig} \cdot \mathsf{Real\text{-}PKI} \leq^{I_{\mathsf{PKIEnc}}} \mathsf{Ideal\text{-}PKISig}$.

The proof of this theorem is again highly modular and leverages such properties of the realization relation as the composition theorem, reflexivity, and transitivity. The basic structure of the proof is analogous to the one for public-key encryption. We split Ideal-PKISig and Real-PKISig into two parts: i) signing and verification and ii) key registration and retrieving of verification keys. The most involved part is to show that the real component for signing and verification realizes the corresponding ideal component. Here we make use of an existing results in the cryptographic literature, in particular [26], and reduce the statement to a corresponding statement in the Turing machine model. We refer to the extended version of this paper [22] for details.

5 Private Symmetric Encryption

In this section, we present an ideal functionality for what we call *private symmetric encryption* and a realization of this functionality. Private symmetric encryption allows a user to encrypt messages (using a symmetric encryption scheme) just for herself. She does not share the symmetric key with other parties. This is useful, for example, to store confidential information on an untrusted medium. Since keys do not have to be shared between parties, the functionality can be kept quite simple.

The public interface I_{SymEnc} of this functionality and its realization consists of only one class `SymEnc` with two methods: `encrypt` and `decrypt`. These methods use a symmetric key generated when an object of this class is created.

In the ideal functionality Ideal-SymEnc for private symmetric encryption, encryption and decryption work analogously to the case of public-key encryption: a sequence of zeros is encrypted instead of the given plaintext and the ciphertext obtained in this way is logged along with the plaintext, which enables the functionality to recover this plaintext when the ciphertext is to be decrypted. The realization Real-SymEnc simply uses the encapsulated key to encrypt and decrypt messages using a symmetric encryption scheme. Clearly, there is no need to model (static) corruption here: a dishonest party can simply perform private symmetric encryption by himself. We refer the reader to the extended version of this paper [22], as well as [23] for the full code of Ideal-SymEnc and Real-SymEnc.

We obtain the following result. We omit the proof here because it closely follows the one for public-key encryption only that it is much simpler now, as we neither need to consider a public-key infrastructure nor corruption (see [21] for a corresponding result in a Turing machine model).

Theorem 6. *If* Real-SymEnc *uses an IND-CCA2-secure symmetric encryption scheme, then* Real-SymEnc $\leq^{I_{PKIEnc}}$ Ideal-SymEnc.

6 Nonce Generation

In this section, we propose an ideal functionality and its realization for nonce generation, formulated in Java (Jinja+). The property that the ideal functionality is supposed to provide is nonce freshness, i.e., nonces returned by the functionality should always be different to the nonce that have been returned so far (no collisions); unguessability of nonces is not intended to be modeled by this functionality.

The public interface I_{Nonce} for this functionality consists of one class `NonceGen` with one method `newNonce` only, which is supposed to return a fresh nonce.

The ideal functionality Ideal-Nonce for nonce generation works as follows. The functionality maintains an, initially empty, collection (formally, a static list) of nonces that have been returned so far. When the method `newNonce` is called, the environment/simulator is asked to provide a bitstring; more precisely, the method `CryptoLib.newNonce()`, which is supposed to be provided by the environment is called. Then, the method `newNonce` checks whether the returned bitstring is fresh, i.e., whether it does not already belong to the collection of returned nonces. If the nonce is indeed fresh, the nonce is added to the collection and returned to the caller of the method. Otherwise, the above process is repeated until a fresh nonce is returned by the environment/simulator. This guarantees that Ideal-Nonce always outputs a fresh nonce.

In the realization Real-Nonce of Ideal-Nonce, if the method `newNonce` is called, a bitstring of the length of the security parameter is picked uniformly at random and then returned to the caller. More precisely, we assume the method `CryptoLib.newNonce()` called by Real-Nonce to work in this way.

We refer the reader to the extended version [22] for the most important part of the code of Ideal-Nonce and Real-Nonce; see [23] for the full code. Now, it is easy to prove that Real-Nonce realizes Ideal-Nonce.

Theorem 7. Real-Nonce $\leq^{I_{\mathsf{Nonce}}}$ Ideal-Nonce.

To prove this theorem, we let the simulator S work just like Real-Nonce, i.e., when asked to provide a new nonce by Ideal-Nonce, it picks a bitstring of the length of the security parameter uniformly at random and returns this bitstring to Ideal-Nonce. Now, Real-Nonce cannot be distinguished by any (polynomial bounded) environment from $S \cdot$ Ideal-Nonce unless Real-Nonce produces a collision, which, however, happens with negligible probability only.

7 The Case Study

As a case study of the results obtained in this paper, we now describe the verification of a cloud storage system implemented in Java. This system illustrates how the ideal functionalities we have developed and presented in this paper can be used to analyze an interesting and non-trivial Java program. As already mentioned in the introduction, except for the work in [24], where only a much simpler Java program has been considered, there has been no other work on establishing cryptographic (indistinguishability) properties for Java programs.

In what follows, we first provide a brief description of the cloud storage system program. Then we state the (cryptographic) security property that we verify and, finally, report on the verification process carried out using the tool Joana [16, 17], which, as already mentioned, allows for the fully automatic verification of noninterference properties of Java programs.

Description of the Cloud Storage System. We have implemented a cloud storage system that allows a user (through her client application) to store data on a remote server such that confidentiality of the data stored on the server is guaranteed even if the server is untrusted: data stored on the server is encrypted using a symmetric key known only to the client.

More specifically, data is stored (encrypted with the symmetric key of a user) on the server along with a label and a counter (a version number). When data is to be stored under some label, a new (higher) counter is chosen and the data is stored under the label and the new counter; old data is still preserved (under smaller counters). Different users can have data repositories on one server. These repositories are strictly separated. The system can be used to securely store any kind of data. A user may use our cloud storage system, for example, to store her passwords remotely on a server such that she has access to them on different devices.

Communication between a client and a server is secured and authenticated using functionalities for public-key encryption and digital signatures. Moreover, the functionality for nonce generation is essential to prevent replay attacks (when the client and the server run a sub-protocol to synchronize counter values for labels). The extended version of this paper [22] gives a more detailed description of our application; see [23] for the full code of the system.

The Security Property. As mentioned, the most fundamental security property of the cloud storage system is confidentiality of the stored data. This property is supposed to be guaranteed even if the server and all clients of other users may be dishonest and cooperate with an active adversary.

To formulate this confidentiality property, we provide (besides the code of the client and the server) a setup class with the method `main`, which gets a secret bit `secret_bit` as input. This method models the interaction between the program of an honest client and the active adversary (the environment). The adversary has full control over the network and subsumes the server and all dishonest clients. The adversary also controls the actions taken by the honest client. In particular, he determines the label and data items the honest client is supposed to store on the server. More precisely, in every request, the adversary provides a pair of data items. The secret bit `secret_bit` determines which of the two items the client actually asks the server to store (see [22] for a more detailed explanation of the setup class and [23] for the full code).

The security property now requires that no (probabilistic polynomial-time) adversary should be able to determine the secret bit `secret_bit`, and hence, whether the data items in the first or in the second component of the item pairs provided by the adversary are sent by the client. This specifies a strong cryptographic privacy property, common in cryptography. Formally, this indistinguishability property is state as follows:

$$\mathsf{CS}_R[\mathtt{false}] \approx^{\emptyset}_{\mathsf{comp}} \mathsf{CS}_R[\mathtt{true}] \tag{1}$$

where $\mathsf{CS}_R[b]$ denotes the described system, consisting of the setup class and the client class, with `secret_bit` set to b. The index R indicates that in this system the cryptographic operations are carried out using the real cryptographic schemes (rather than ideal functionalities).

We note that the computational indistinguishability relation in (1) uses the empty interface $I = \emptyset$. This means that the adversary (environment) cannot directly call methods of the client object. As explained before, by the definition of the setup class, the environment can nonetheless determine which actions are taken and when. We also point out that CS_R is an open system which uses some classes not defined within CS_R, such as a network library. These classes are provided by the environment and, therefore, are untrusted. Thus, property (1) implies confidentiality of the stored messages no matter how such untrusted libraries are implemented.

Verification of the Security Property. In order to prove (1), by Theorem 3 it suffices to show that

$$\mathsf{CS}_I[b] \text{ is } I\text{-noninterferent,} \tag{2}$$

where CS_I denotes the system which coincides with CS_R except that the real cryptographic schemes are replaced by their ideal counterparts (ideal functionalities), i.e., Ideal-PKEnc, Ideal-Sig, Ideal-SymEnc, and Ideal-Nonce. Since, as can easily been seen, $\mathsf{CS}_I[b]$ satisfies the conditions of Theorem 2, we can further reduce checking (2) to checking the following property:

$$\tilde{E}_{\vec{u}} \cdot \mathsf{CS}_I[b] \quad \text{is noninterferent for all } \vec{u}, \tag{3}$$

where the family of systems $\tilde{E}_{\vec{u}}$, parameterized by a finite sequence of integers $\vec{u}$, is as described in Section 2. This system can be automatically generated from $\mathsf{CS}_I[b]$. Also note that by "noninterference" we mean standard termination-insensitive noninterference (see Section 2). Altogether it suffices to prove (3) in order to obtain (1).

Joana was easily able to establish property (3). It took about 17 seconds on a standard PC (Core i5 2.3GHz, 8GB RAM) to finish the analysis of the program (with a size of 950 LoC). Note that the actual running code of the distributed system is much bigger than what Joana needed to analyze, because the code of the distributed system includes untrusted libraries, such as the standard Java library for networking, which do not need to be analyzed, as already mentioned above.

Acknowledgment. This work was partially supported by *Deutsche Forschungsgemeinschaft* (DFG) under Grant KU 1434/6-2 within the priority programme 1496 "Reliably Secure Software Systems – RS^3".

References

1. Aizatulin, M., Gordon, A.D., Jürjens, J.: Extracting and verifying cryptographic models from C protocol code by symbolic execution. In: Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011). ACM (2011)
2. Aizatulin, M., Gordon, A.D., Jürjens, J.: Computational verification of C protocol implementations by symbolic execution. In: ACM Conference on Computer and Communications Security (CCS 2012). ACM (2012)
3. Asokan, N., Niemi, V., Laitinen, P.: On the Usefulness of Proof-of-Possession. In: Proceedings of the 2nd Annual PKI Research Workshop (2003)
4. Bellare, M., Desai, A., Pointcheval, D., Rogaway, P.: Relations among Notions of Security for Public-Key Encryption Schemes. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 26–45. Springer, Heidelberg (1998)
5. Bhargavan, K., Fournet, C., Gordon, A.D.: Modular verification of security protocol code by typing. In: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010). ACM (2010)
6. Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.-Y.: Implementing TLS with verified cryptographic security. In: IEEE Symposium on Security & Privacy, Oakland (2013)
7. Blanchet, B.: A Computationally Sound Mechanized Prover for Security Protocols. In: IEEE Symposium on Security and Privacy (S&P 2006). IEEE Computer Society (2006)
8. Cadé, D., Blanchet, B.: Proved Generation of Implementations from Computationally Secure Protocol Specifications. In: Basin, D., Mitchell, J.C. (eds.) POST 2013. LNCS, vol. 7796, pp. 63–82. Springer, Heidelberg (2013)
9. Canetti, R.: Universally Composable Security: A New Paradigm for Cryptographic Protocols. In: Proceedings of the 42nd Annual Symposium on Foundations of Computer Science (FOCS 2001). IEEE Computer Society (2001)
10. Canetti, R.: Universally Composable Signature, Certification, and Authentication. In: Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW-17 2004). IEEE Computer Society (2004)
11. Chaki, S., Datta, A.: ASPIER: An automated framework for verifying security protocol implementations. In: Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF 2009). IEEE Computer Society (2009)
12. Fournet, C., Kohlweiss, M., Strub, P.-Y.: Modular code-based cryptographic verification. In: Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011). ACM (2011)

13. Goguen, J.A., Meseguer, J.: Security Policies and Security Models. In: Proceedings of IEEE Symposium on Security and Privacy (1982)
14. Goldwasser, S., Micali, S., Rivest, R.L.: A digital signature scheme secure against adaptive chosen-message attacks. SIAM Journal on Computing 17(2), 281–308 (1988)
15. Goubault-Larrecq, J., Parrennes, F.: Cryptographic protocol analysis on real C code. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 363–379. Springer, Heidelberg (2005)
16. Graf, J., Hecker, M., Mohr, M.: Using JOANA for Information Flow Control in Java Programs - A Practical Guide. In: Proceedings of the 6th Working Conference on Programming Languages (ATPS 2013). Lecture Notes in Informatics (LNI), vol. 215. Springer, Heidelberg (2013)
17. Hammer, C., Snelting, G.: Flow-Sensitive, Context-Sensitive, and Object-sensitive Information Flow Control Based on Program Dependence Graphs. International Journal of Information Security 8(6), 399–422 (2009)
18. Klein, G., Nipkow, T.: A Machine-Checked Model for a Java-Like Language, Virtual Machine, and Compiler. ACM Trans. Program. Lang. Syst. 28(4), 619–695 (2006)
19. Küsters, R.: Simulation-Based Security with Inexhaustible Interactive Turing Machines. In: Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW-19 2006). IEEE Computer Society (2006)
20. Küsters, R., Tuengerthal, M.: Joint State Theorems for Public-Key Encryption and Digital Signature Functionalities with Local Computation. Technical Report 2008/006, Cryptology ePrint Archive (2008), `http://eprint.iacr.org/2008/006`
21. Küsters, R., Tuengerthal, M.: Universally Composable Symmetric Encryption. In: Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF 2009). IEEE Computer Society (2009)
22. Küsters, R., Scapin, E., Truderung, T., Graf, J.: Extending and Applying a Framework for the Cryptographic Verification of Java Programs. Cryptology ePrint Archive, Report 2014/038 (2014), `http://eprint.iacr.org/2014/038`
23. Küsters, R., Scapin, E., Truderung, T., Graf, J.: A Java Implementation of a Cloud Storage System (2013), `http://infsec.uni-trier.de/publications/software/CloudStorage.zip`
24. Küsters, R., Truderung, T., Graf, J.: A Framework for the Cryptographic Verification of Java-like Programs. In: IEEE Computer Security Foundations Symposium, CSF 2012. IEEE Computer Society (2012)
25. Küsters, R., Truderung, T., Graf, J.: A Framework for the Cryptographic Verification of Java-like Programs. Cryptology ePrint Archive, Report 2012/153 (2012), `http://eprint.iacr.org/2012/153`
26. Küsters, R., Tuengerthal, M.: Joint State Theorems for Public-Key Encryption and Digital Signature Functionalities with Local Computation. In: Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF 2008). IEEE Computer Society (2008)
27. Küsters, R., Tuengerthal, M.: The IITM Model: a Simple and Expressive Model for Universal Composability. Technical Report 2013/025, Cryptology ePrint Archive (2013), `http://eprint.iacr.org/2013/025`
28. Nipkow, T., von Oheimb, D.: Java$_{light}$ is Type-Safe — Definitely. In: POPL (1998)
29. Pfitzmann, B., Waidner, M.: A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission. In: IEEE Symposium on Security and Privacy. EEE Computer Society (2001)