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 [10, 20, 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.

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., [5]). 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., [10, 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 [17, 18], 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., [6, 12, 16]). 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, 4]), ii) they derive formal models from the source code and analyze these models using specialized tools for cryptographic verification, such as the tool CryptoVerif [8] (see, e.g., [2]), or iii) they derive source code from formal specifications (see, e.g., [9]). 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. [7, 13]. 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 appendix.

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* [19] 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.[3]

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 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.*

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

Noninterference. The (standard) noninterference notion for confidentiality [14] 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 $\tilde{E}_{\vec{u}}^{I,I_E}$ (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 [17, 18], 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 ($\forall\ \vec{a}_1, \vec{a}_2$: $P[\vec{a}_1] \cdot R \approx^{I}_{\mathsf{comp}} P[\vec{a}_2] \cdot R$) 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.[4] 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}.

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

[4] 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.

```
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 Appendix D.1; see [23] for the full code.

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:

```
38 public class PKI {
39   static void register(int id, byte[] domain, byte[] pubKey)
40                                             throws PKIError, NetworkError;
41   static byte[] getKey(int id, byte[] domain)
42                                             throws PKIError, NetworkError;
43 }
```

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 Appendix D.2; 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 [5] 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., [10, 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 Appendix D.5). 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 Appendix B. 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., [11, 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`:

```
23 try {
24   Verifier verifier = RegisterSig.getVerifier(ID_A, PKI_DOMAIN);
25   verifier.verify(signature, message);
26 } catch(PKIError e) {}     // id has not been successfully registered
27   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:

```
28 Verifier ver = new Verifier(verif_key);
29 try {
30     RegisterSig.registerVerifier(ID, ver, PKI_DOMAIN);
31 } 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 $I_{\text{CryptoLibSig}}$ which contains methods for key generation, signing, and verification. Analogously to the interface $I_{\text{CryptoLibEnc}}$ 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 $I_{\text{CryptoLibSig}}$. 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 $I_{\text{CryptoLibSig}}$ 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 $I_{\text{CryptoLibSig}}$ 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 $I_{\text{CryptoLibSig}}$.

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 Appendix D.3; 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 Appendix D.4; 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 [15] 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* Real-PKI$\leq^{I_{PKI}}$ Ideal-PKI*, then* Real-PKISig $\cdot$ Real-PKI $\leq^{I_{PKIEnc}}$ Ideal-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 Appendix C 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 Appendices D.6 and D.7 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 [22] 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 Appendices D.8 and D.9 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 $\leq^{I_{\text{Nonce}}}$ 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 [17, 18], 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). Appendix A.1 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 Appendix A.2 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. Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. 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. Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. Computational verification of C protocol implementations by symbolic execution. In *ACM Conference on Computer and Communications Security (CCS 2012)*. ACM, 2012.
3. N. Asokan, Valtteri Niemi, and Pekka Laitinen. On the Usefulness of Proof-of-Possession. In *Proceedings of the 2nd Annual PKI Research Workshop*, 2003.
4. Michael Backes, Matteo Maffei, and Dominique Unruh. Computationally sound verification of source code. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS 2010)*. ACM, 2010.
5. M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations Among Notions of Security for Public-Key Encryption Schemes. In *Advances in Cryptology, 18th Annual International Cryptology Conference (CRYPTO 1998)*, volume 1462 of *LNCS*. Springer, 1998.
6. Karthikeyan Bhargavan, Cédric Fournet, and Andrew D. Gordon. 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.
7. Karthikeyan Bhargavan, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, and Pierre-Yves Strub. Implementing TLS with verified cryptographic security. In *IEEE Symposium on Security & Privacy (Oakland)*, 2013.
8. B. Blanchet. A Computationally Sound Mechanized Prover for Security Protocols. In *IEEE Symposium on Security and Privacy (S&P 2006)*. IEEE Computer Society, 2006.
9. David Cadé and Bruno Blanchet. Proved Generation of Implementations from Computationally Secure Protocol Specifications. In *Principles of Security and Trust - Second International Conference (POST 2013)*, volume 7796 of *LNCS*. Springer, 2013.
10. R. Canetti. 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.
11. R. Canetti. Universally Composable Signature, Certification, and Authentication. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW-17 2004)*. IEEE Computer Society, 2004.
12. 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)*. IEEE Computer Society, 2009.
13. Cédric Fournet, Markulf Kohlweiss, and Pierre-Yves Strub. Modular code-based cryptographic verification. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS 2011)*. ACM, 2011.
14. Joseph A. Goguen and José Meseguer. Security Policies and Security Models. In *Proceedings of IEEE Symposium on Security and Privacy*, 1982.

15. S. Goldwasser, S. Micali, and R.L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):281–308, 1988.
16. 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. Springer, 2005.
17. Jürgen Graf, Martin Hecker, and Martin Mohr. Using JOANA for Information Flow Control in Java Programs - A Practical Guide. In *Proceedings of the 6th Working Conference on Programming Languages (ATPS'13)*, Lecture Notes in Informatics (LNI) 215. Springer Berlin / Heidelberg, February 2013.
18. 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.
19. 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.
20. R. Küsters. 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.
21. R. Küsters and M. Tuengerthal. Joint State Theorems for Public-Key Encryption and Digital Signature Functionalities with Local Computation. Technical Report 2008/006, Cryptology ePrint Archive, 2008. Available at http://eprint.iacr.org/2008/006.
22. R. Küsters and M. Tuengerthal. Universally Composable Symmetric Encryption. In *Proceedings of the 22nd IEEE Computer Security Foundations Symposium (CSF 2009)*. IEEE Computer Society, 2009.
23. Ralf Küsters, Enrico Scapin, and Tomasz Truderung. A Java Implementation of a Cloud Storage System, 2013. http://infsec.uni-trier.de/publications/software/CloudStorage.zip.
24. Ralf Küsters, Tomasz Truderung, and Jürgen Graf. A Framework for the Cryptographic Verification of Java-like Programs. In *IEEE Computer Security Foundations Symposium, CSF 2012*. IEEE Computer Society, 2012.
25. Ralf Küsters, Tomasz Truderung, and Jürgen Graf. A Framework for the Cryptographic Verification of Java-like Programs. Cryptology ePrint Archive, Report 2012/153, 2012. http://eprint.iacr.org/2012/153.
26. Ralf Küsters and Max Tuengerthal. 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. Ralf Küsters and Max Tuengerthal. The IITM Model: a Simple and Expressive Model for Universal Composability. Technical Report 2013/025, Cryptology ePrint Archive, 2013. Available at http://eprint.iacr.org/2013/025.
28. Tobias Nipkow and David von Oheimb. Java$_{light}$ is Type-Safe — Definitely. In *POPL 1998*.
29. 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*. IEEE Computer Society, 2001.

A The Case Study

A.1 Description of the Cloud Storage System

Store

(1) $C \to S:\ Enc_S(userID, Sig_C[\texttt{STORE}, label, counter, SymEnc_k(message)])$
(2a) $S \to C:\ Enc_C(Sig_S[Sign_C, \texttt{STORE\_OK}])$
(2b) $S \to C:\ Enc_C(Sig_S[Sign_C, \texttt{STORE\_FAIL}, lastCounter])$

Retrieve

(3) $C \to S:\ Enc_S(userID, Sig_C[\texttt{RETRIEVE}, label, counter])$
(4a) $S \to C:\ Enc_C(Sig_S[Sign_C, \texttt{RETRIEVE\_OK}, encryptedMsg, authToken])$
(4b) $S \to C:\ Enc_C(Sig_S[Sign_C, \texttt{RETRIEVE\_FAIL}])$

Synchronization

(5) $C \to S:\ Enc_S(userID, Sig_C[\texttt{GET\_COUNTER}, label, nonce])$
(6) $S \to C:\ Enc_C(Sig_S[Sign_C, \texttt{LAST\_COUNTER}, serverCounter, nonce])$

Fig. 1. Messages exchanged between the client and the server, where $Enc_S(m)$ and $Enc_C(m)$ denote m encrypted under the public key of the server and the client, respectively. Analogously, $Sig_S[m]$ and $Sig_C[m]$ denote the signatures of the server and the client, respectively, on m, along with the message m itself. Finally, $SymEnc_k(m)$ denotes m encrypted under the symmetric key k. By $Sign_C$, we denote the signature (not the signed message) of C in the previous message. For example, in (2a) $Sign_C$ denotes C's signature in message (1).

In our system, data is stored (encrypted) 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.

When created, client and server objects are provided with all necessary key material. In particular, a client object is provided with a user ID and the corresponding public and private encryption and signing keys as well as the symmetric key for encrypting data. The server obtains its public and private encryption and signing keys.

The client class of our system offers two methods: `store` (with parameters *message* (data) and *label*), to store data (*message*) under a chosen label (*label*), and `retrieve` (with the parameter *label*), to retrieve data stored under a label (*label*). The client and the server internally maintain the current counter. A counter recorded on the client for a label may differ from the one recorded on the server since, for example, another instance of the client (with the same user ID) may have stored further data on the server meanwhile. Store and retrieve actions therefore always start with a synchronization step (see Figure 1, (5) and (6)) where the client asks the server for the current counter for the considered label. If this value is higher than the one stored locally by the client, the client updates

its counter to this higher value. If the value is lower, the client throws an exception. The nonce in messages (5) and (6) is used to prevent replay attacks.

Now, when the method `store` is invoked with parameters *message* and *label*, the client object, after having synchronized the counter with the server (see above), sends message (1) in Figure 1 to the server, where *counter* is the current value of the counter for *label* obtained after synchronization and k is a private key of the client (not shared with any other party). The client's signature in (1) is stored by the server along with *label*, *counter*, and the ciphertext $SymEnc_k(message)$, and is used later as an authentication token (when retrieving the data). The server may reply with an error message (2b), indicating a counter error (some message has already been stored for the given combination of *counter* and *label*). Otherwise, the server acknowledges that the storage operation was successful (2a).

When the method `retrieve` is called with parameter *label*, the client sends, again after synchronization with the server, message (3) to the server, where *counter* is the current value of the counter for *label* after synchronization. The server can, again, respond with an error message (4b) (indicating that there is no message stored under the given combination *label*/*counter* for that user), or it responds with the message (4a), containing the encrypted data *encryptedMsg* stored under *label*/*counter* and an authentication token *authToken* (see above), which proves to the client that the response of the server is correct.

The full code of the system is available under [23].

A.2 Description of the Setup Program

The setup program takes the parameter `secret_bit` of type `boolean` as its input. This program, first, creates a client (i.e. an object of class `Client`) and registers her public-key encryption and signing keys in the public-key infrastructure. If this registration process succeeds, the setup program enters its main loop where the adversary (the environment) determines, one by one, the actions to be taken by the system by sending instructions to `main`. Except for the first instruction, the following instructions can be send by the adversary arbitrarily often.

- The adversary can decide to end the loop by sending a special end instruction.
- The adversary can register a corrupted encryptor and/or a corrupted verifier. In particular, he can register such objects under the fixed identifier of the server. By this, the adversary is able to fully subsume (impersonate) the server: he can decrypt messages encrypted for the server and produce signatures of the server. Analogously, the adversary can register (and then subsume) dishonest clients. Note that the adversary cannot register keys under the ID of the honest client created at the beginning of the setup, because this ID is already taken.
- The adversary can pick an arbitrary label and an arbitrary message to be stored by the honest client on the server, by calling `client.store(label,message)`. More precisely, to do so, the adversary, besides the label, provides a pair (m_0, m_1) of messages of the same length (if the two messages do not have the same length, this step is aborted). Then, depending on the value of the secret bit (`secret_bit`) m_0 or m_1 is picked as `message` (the message to be stored).

– The adversary can choose to have the honest client retrieve a message for a given label (again, determined by the adversary), using `client.retrieve(label)`. The value of the returned message is then ignored. However, notice that this step, according to the honest client program, triggers an exchange of messages between the client and the server (adversary), which includes the encrypted message.

By this, the adversary has full control over the network, the server, all dishonest clients, and over the actions taken by the honest client.

B Proof of Theorem 4

As we have mentioned in Section 3.4, the proof of Theorem 4 is highly modular and leverages such properties of the realization relation as the composition theorem, reflexivity, and transitivity. Due to this modular proof technique, we can even make use of the result proved in [24] for the public-key functionality without corruption and without a PKI.

First, we observer that the ideal functionality Ideal-PKIEnc can be split in the following way:

$$\mathsf{Ideal\text{-}PKIEnc} = \mathsf{Ideal\text{-}PKEnc} \cdot \mathsf{Ideal\text{-}RegEnc} \ ,$$

where Ideal-PKEnc and Ideal-RegEnc are defined as follows:

Ideal-PKEnc consists of the classes `Encryptor` and `Decryptor` of Ideal-PKIEnc, as introduced in Section 3.2. Let I_{PKEnc} denote the public interface of these classes. That is, I_{PKEnc} coincides with I_{PKIEnc}, as defined in Section 3.1, excluding the interface of the class `RegisterEnc`.

Ideal-RegEnc consists of the class `RegisterEnc` of Ideal-PKIEnc. Let I_{EncPKI} denotes the public interface of this class. That is, the interface I_{PKIEnc}, as defined in Section 3.1, restricted to the public interface of the class `RegisterEnc`.

Similarly, Real-PKIEnc can be split in the following way:

$$\mathsf{Real\text{-}PKIEnc} = \mathsf{Real\text{-}PKEnc} \cdot \mathsf{Real\text{-}RegEnc} \ ,$$

where Real-PKEnc and Real-RegEnc are defined as follows:

Real-PKEnc consists of the classes `Encryptor` and `Decryptor` of Real-PKIEnc, as introduced in Section 3.3. Note that the public interface of Real-PKEnc is I_{PKIEnc}, and hence, it is the same as the one for Ideal-PKEnc.

Real-RegEnc consists of the `RegisterEnc` of Real-PKIEnc. Note that the public interface of Real-RegEnc is I_{EncPKI}, and hence, it is the same as the one for Ideal-RegEnc.

Now, we prove the following sequence of realization relationships:

$$\begin{aligned}
\mathsf{Real\text{-}PKIEnc} \cdot \mathsf{Real\text{-}PKI} &= \mathsf{Real\text{-}PKEnc} \cdot \mathsf{Real\text{-}RegEnc} \cdot \mathsf{Real\text{-}PKI} \\
&\leq^{I_{PKIEnc}} \mathsf{Real\text{-}PKEnc} \cdot \mathsf{Real\text{-}RegEnc} \cdot \mathsf{Ideal\text{-}PKI} && (4) \\
&\leq^{I_{PKIEnc}} \mathsf{Real\text{-}PKEnc} \cdot \mathsf{Ideal\text{-}RegEnc} && (5) \\
&\leq^{I_{PKIEnc}} \mathsf{Ideal\text{-}PKEnc} \cdot \mathsf{Ideal\text{-}RegEnc} = \mathsf{Ideal\text{-}PKIEnc} && (6)
\end{aligned}$$

From this, by transitivity of the realization relation, Theorem 4 follows. Now, we establish each of the above relationships.

Lemma 1. *Relationship* (4) *holds true, that is*

$$\mathsf{Real\text{-}PKEnc} \cdot \mathsf{Real\text{-}RegEnc} \cdot \mathsf{Real\text{-}PKI} \leq^{I_{PKIEnc}} \mathsf{Real\text{-}PKEnc} \cdot \mathsf{Real\text{-}RegEnc} \cdot \mathsf{Ideal\text{-}PKI} \ .$$

Proof. This relationship easily follows from the assumption that $\mathsf{Real\text{-}PKI} \leq^{I_{PKI}} \mathsf{Ideal\text{-}PKI}$, the composition theorem, and reflexivity of the realization relation. Indeed, by reflexivity of realization relation, we have that

$$\mathsf{Real\text{-}PKEnc} \cdot \mathsf{Real\text{-}RegEnc} \leq^{I_{PKIEnc}} \mathsf{Real\text{-}PKEnc} \cdot \mathsf{Real\text{-}RegEnc}.$$

Note that $I_{PKIEnc} = I_{PKEnc} \cup I_{EncPKI}$. Together with $\mathsf{Real\text{-}PKI} \leq^{I_{PKI}} \mathsf{Ideal\text{-}PKI}$, by the composition theorem we immediately obtain that

$$\mathsf{Real\text{-}PKEnc} \cdot \mathsf{Real\text{-}RegEnc} \cdot \mathsf{Real\text{-}PKI} \leq^{I_{PKIEnc} \cup I_{PKI}} \mathsf{Real\text{-}PKEnc} \cdot \mathsf{Real\text{-}RegEnc} \cdot \mathsf{Ideal\text{-}PKI}$$

which implies (4), since if a relationship holds for one interface, then also for all of its subinterfaces. □

Lemma 2. *The relationship* (5) *holds true, that is*

$$\mathsf{Real\text{-}PKEnc} \cdot \mathsf{Real\text{-}RegEnc} \cdot \mathsf{Ideal\text{-}PKI} \leq^{I_{PKIEnc}} \mathsf{Real\text{-}PKEnc} \cdot \mathsf{Ideal\text{-}RegEnc}.$$

The two systems are very similar: the main difference is that in the ideal system (the one on the right hand-side) encryptors are stored directly (in a collection of registered encryptors), while in the real system (the one on the left hand-side) public keys are stored instead with wrapping/unwrapping of public keys in encryptors when necessary. Therefore, the relationship holds true even if unbounded environments try to distinguish the two systems. The proof of this lemma is given in Appendix B.1.

Finally, relationship (6) follows immediately by the following fact and again using the composition theorem and reflexivity of realization relation, similarly to the proof of Lemma 1.

Lemma 3. $\mathsf{Real\text{-}PKEnc} \leq^{I_{PKEnc}} \mathsf{Ideal\text{-}PKEnc}$

We prove Lemma 3 by reducing it to the result from [24], where similar functionalities, but without corruption are considered. In the proof we use the fact that corrupted encryptors can be simulated directly by an environment. The proof is given in Appendix B.2.

B.1 Proof of Lemma 2

In this section, we prove that

$$\mathsf{Real\text{-}PKEnc} \cdot \mathsf{Real\text{-}RegEnc} \cdot \mathsf{Ideal\text{-}PKI} \leq^{I_{PKIEnc}} \mathsf{Real\text{-}PKEnc} \cdot \mathsf{Ideal\text{-}RegEnc}.$$

In order to do this, we need to show that there exists a simulator *Sim* such that

$$S = \mathsf{Real\text{-}PKEnc} \cdot \mathsf{Real\text{-}RegEnc} \cdot \mathsf{Ideal\text{-}PKI} \approx^{I_{PKIEnc}}_{\mathsf{comp}} \mathit{Sim} \cdot \mathsf{Real\text{-}PKEnc} \cdot \mathsf{Ideal\text{-}RegEnc} = \tilde{S}$$

Let *Sim* be the simple forwarding simulator that translates calls to the simulator interface of Ideal-RegEnc (that is class `RegisterEncSim`) into calls to the simulator interface of Ideal-PKI (class `PKISim`):

```
public class RegisterEncSim {
  public static boolean register(int id, byte[] domain, byte[] publicKey) {
    return PKISim.register(id, domain, publicKey);
  }
  public static boolean getEncryptor(int id, byte[] domain) {
    return PKISim.getKey(id, domain);
  }
}
```

We are able to prove the stronger property

$$S \approx_{\mathsf{perf}}^{I_{PKIEnc}} \tilde{S}$$

that is, the two systems are perfectly indistinguishable, that is indistinguishable by an unbounded, deterministic adversary (see [24] for details). For this, let us take an arbitrary deterministic I_{PKIEnc}-environment E for S (and hence for $\tilde{S}$). To complete the proof, it remains to show that

$$E \cdot S \equiv_{\mathsf{perf}} E \cdot \tilde{S} \tag{7}$$

which simply means that the environment E in the runs of both $E \cdot S$ and $E \cdot \tilde{S}$ outputs the same value.

On the intuitive level the above statement is quite straightforward. Indeed, the systems S and $\tilde{S}$, from the point of view of any environment E, realize very similar computations with the only difference being how they implement registration of encryptors. In the system S, public keys are kept in a collection (along with user identifiers); those keys are retrieved from an encryptor by method `registerEncryptor` and, conversely, wrapped into a newly created encryptor by method `getEncryptor`. In the system $\tilde{S}$, on the other hand side, encryptors are stored directly (along with user identifiers). Method `registerEncryptor` simply adds such an encryptor (along with a user identifier) into a collection, when method `getEncryptor` retrieves an appropriate encryptor and returns a newly created copy of it. These computations produce the same result, up to the internal state of the component $S/\tilde{S}$. In other words, the state of the computations in the considered systems is the same from the point of view of the environment and so, in particular, the value of variable `result` (which is determined by the environment) at the end of the runs is the same in both systems.

To formalize the intuitive argument given above, we need to introduce some notation.

Structure of states in a run. A configuration q of Jinja+, as defined in [24], is of the form (e,s), where e is a Jinja+ expression and s is a state. A *state* is a pair (h,l) of a heap and a store. A *heap* is a mapping from references (addresses) to object instances and a *store* is a mapping from variable names to values. A value can be either a reference or a value of a primitive type.

One particular type of expression is a *block expression* of the form $\{V : T;\ e\}_C$ or $\{V : T;\ V := \texttt{Val}\ v;\ e\}_C$, where V is a local variable (whose scope is this block) of type T and, in the second variant, with value `Val` v, e is an expression (e can access the local variable V), and C is a class name (denoting that the block originates from the code of the class C).

In general, an expression can contain many blocks as its subexpression. However, when we study expressions that occur in actual runs, it turns out that they have a simpler

form, where all blocks are located on one path. Formally, let

$$q_0 = \langle e_0, \langle h_0, l_0 \rangle \rangle \xrightarrow{\ell} \langle e_1, \langle h_1, l_1 \rangle \rangle \xrightarrow{\ell} \cdots$$

be a run with the initial state $s_0 = \langle h_0, l_0 \rangle$. By the definition of the initial state [24, 25], h_0 is empty and l_0 bounds the static variables of the program to their initial values (and no other variables). By inspecting the rules of Jinja+ (see Appendix E), one can see that, for every $i = 0, 1, \ldots$,

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

The definitions and results given below assume that expressions originate from runs of Jinja+ systems and therefore they are of the above form.

By $\mathfrak{C}[\cdot]$ we denote an expression context (that is, an expression with a hole) and by $\mathfrak{C}[e]$ we denote the expression obtained by replacing the hole by e.

We can now state the two following lemmas.

Lemma 4. *$\langle e, (h,l) \rangle \xrightarrow{-} \langle e', (h',l') \rangle$ if and only if e contains no block.*

The proof can be easily done by structural induction, considering all possible rules of Jinja+ (Appendix E) that produce a reduction step with label $-$.

Lemma 5. *If $\langle e, (h,l) \rangle \xrightarrow{D} \langle e', (h',l') \rangle$, then*

- *e is of the form $\mathfrak{C}[e_0]$, where e_0 is a block expression of class D without blocks;*
- *$\langle e_0, (h,l) \rangle \xrightarrow{D} \langle e'_0, (h',l') \rangle$;*
- *$e' = \mathfrak{C}[e'_0]$.*

Again, this lemma can be easily proven by structural induction.

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

- if e contains no block, then $\mathrm{sub}_{\mathscr{C}}(e) = e$,
- if e is not a block, but contains one, that is $e = E[b]$, then $\mathrm{sub}_{\mathscr{C}}(e) = E[\mathrm{sub}_{\mathscr{C}}(b)]$,
- if $e = \{V : T;\ e'\}_D$ with $D \in \mathscr{C}$, then $\mathrm{sub}_{\mathscr{C}}(e) = \{V : T;\ \mathrm{sub}_{\mathscr{C}}(e')\}_D$ (and similarly for $e = \{V : T;\ V := \texttt{Val}\ v;\ e'\}_D$),
- if $e = \{V : T;\ e'\}_D$ with $D \notin \mathscr{C}$, and e' contains no blocks, then $\mathrm{sub}_{\mathscr{C}}(e) = \bot$.
- if $e = \{V : T;\ E[b]\}_D$, where b is a block with $D \notin \mathscr{C}$, then $\mathrm{sub}_{\mathscr{C}}(e) = \mathrm{sub}(b)$,

Corresponding states. As it has been already stated, our goal is to show that (7) holds true. The systems S and $\tilde{S}$ we consider in this equivalence share the component Real-PKEnc and it will be useful for the remainder of the proof to use the following notation. Let E' denote $E \cdot \mathsf{Real\text{-}PKEnc}$ (that is, E' is the environment enlarged by the shared functionality Real-PKEnc), let

$$T = \mathsf{Real\text{-}RegEnc} \cdot \mathsf{Ideal\text{-}PKI},$$
$$\tilde{T} = \mathsf{Ideal\text{-}RegEnc} \cdot \mathit{Sim}.$$

Using this notation, (7), that is the equivalence to be proved, can be represented as

$$E' \cdot T \equiv_{\mathsf{perf}} E' \cdot \tilde{T} \tag{8}$$

Let $q = \langle e, (h, l) \rangle$ be a configuration of $E' \cdot T$ and $\tilde{q} = \langle \tilde{e}, (\tilde{h}, \tilde{l}) \rangle$ be a configuration of $E' \cdot \tilde{T}$.

We say that a bijection $f : R_1 \to R_2$, where R_1 and R_2 are subsets of the set of all references, is an $(h, \tilde{h})$-*congruence*, if for all $r, \tilde{r}$ such that $\tilde{r} = f(r)$ one of the following conditions holds true:

(i) Both r and $\tilde{r}$ point to objects of the same class C defined in E' and for every field m of C, either (a) both $r' = h(r).m$ and $\tilde{r}' = \tilde{h}(\tilde{r}).m$ have the same primitive value or (b) both r' and $\tilde{r}'$ are references and $\tilde{r}' = f(r')$.
(ii) Both r and $\tilde{r}$ point to an array of the same type T and the same length l such either (a) T is a primitive type and r and $\tilde{r}$ contain the same values or (b) T is a class and, for every $i \in \{0, \ldots, l-1\}$ and every pair of corresponding references $r' = h(r)[i]$ and $\tilde{r}' = \tilde{h}(\tilde{r})[i]$, we have $\tilde{r}' = f(r')$.

Let f be an $(h, \tilde{h})$-congruence. For primitive values $v, \tilde{v}$, we write $v \equiv_f \tilde{v}$, if simply $v = \tilde{v}$. For references $r, \tilde{r}$, we write $r \equiv_f \tilde{r}$, if $\tilde{r} = f(r)$. Finally, we extend the relation $\equiv_f$ to expressions by the structural isomorphism, that is $e \equiv_f \tilde{e}$ holds if and only if e and $\tilde{e}$ are (syntactically) equal, up to references occurring as their corresponding subexpressions which need to be in the relation $\equiv_f$.

We also define $l \equiv_f \tilde{l}$ to be true if, intuitively, the state of E' (given by static variables of E') is the same up to reference renaming f and the state of T and $\tilde{T}$, although different, represent essentially the same store of registered encryptors. Formally, we put $l \equiv_f \tilde{l}$ if and only if

(a) For every static variable x defined in E' we have $l(V) \equiv_f \tilde{l}(V)$.
(b) Static variable `IdealPKI.entries` (defined in T) and static variable `RegisterEnc.registeredAgents` (defined in $\tilde{T}$) contain information which is strictly corresponding in the following sense.
First let us observe that `IdealPKI.entries` points to a list of entries, each containing `id` of type `int`, and `domain` and `key` of type `byte[]`. Similarly, `RegisterEnc.registeredAgents` points to a list of entries, each containing `id` of type `int`, `domain` of type `byte[]`, and `domain` of type `Encryptor`.
Now, for we require that, for all values *id*, *domain*, and *key*, where *id* is an integer and *domain* and *key* are arrays of bytes, the following equivalence holds: the list

pointed to by `IdealPKI.entries` contains a tuple with values (*id*, *domain*, *key*) if and only if the list pointed to by `RegisterEnc.registeredAgents` contains a tuple with values *id*, *domain* and an encryptor containing *key* as its public key (that is its field `publicKey` points to an array containing the bitstring *key*).

We say that $q = \langle e, (h,l)\rangle$ and $\tilde{q} = \langle \tilde{e}, (\tilde{h},\tilde{l})\rangle$ are *corresponding*, if there exists a $(h,\tilde{h})$-congruence f such that

1. $\mathrm{sub}_{E'}(e) \equiv_f \mathrm{sub}_{E'}(\tilde{e})$,
2. $l \equiv_f \tilde{l}$.

Condition 1 above means that the expressions e and $\tilde{e}$ (representing the code being executed), when stripped off the code originatin in $T/\tilde{T}$, are the same (up to reference renaming). Condition 2 says that the state, as given by static variables, is the same, up to reference renaming an up to (not-essential) differences in how T and $\tilde{T}$ store public keys.

We will sometimes write that q and $\tilde{q}$ are *f-corresponding* to make it explicit which congruence is used.

Lemma 6. *Let $q = \langle e, (h,l)\rangle$ and $\tilde{q} = \langle \tilde{e}, (\tilde{h},\tilde{l})\rangle$ be f-corresponding configurations for an $(h,\tilde{h})$-congruence f. Let $q \xrightarrow{-} q'$ and $\tilde{q} \xrightarrow{-} \tilde{q}'$. Then $q' = \langle e', (h',l')\rangle$ and $\tilde{q}' = \langle \tilde{e}', (\tilde{h}',\tilde{l}')\rangle$ are f'-corresponding for an $(h',\tilde{h}')$-congruence f' which is an extension of f.*

Proof. First of all, we prove that the Jinja+ rules applied to q and to $\tilde{q}$ are indeed the same. By Lemma 4, since $\langle e, (h,l)\rangle \xrightarrow{-} \langle e', (h',l')\rangle$ and $\langle \tilde{e}, (\tilde{h},\tilde{l})\rangle \xrightarrow{-} \langle \tilde{e}', (\tilde{h}',\tilde{l}')\rangle$, e and $\tilde{e}$ contain no blocks. Therefore, by the definition of the pruning operator, we have $\mathrm{sub}_{E'}(e) = e$ and $\mathrm{sub}_{E'}(\tilde{e}) = \tilde{e}$ and hence, since they are f-corresponding, we have $e \equiv_f \tilde{e}$. Moreover, the f-corresponding relation means also $l \equiv_f \tilde{l}$.

The relation $\equiv_f$ between expressions implies that e and $\tilde{e}$ are syntactically equal, up to reference occurring as their corresponding subexpressions. Since applicability of no Jinja+ rule depends on the particular values of references, the same rule is applied to $\langle e, (h,l)\rangle$ and to $\langle \tilde{e}, (\tilde{h},\tilde{l})\rangle$.

We can now prove the q' and $\tilde{q}'$ are also corresponding, depending on the Jinja+ rule applied to both q and $\tilde{q}$.

Rule 21. In this case we have

$$q = \langle \texttt{Cast}\ C\ e_1, (h,l)\rangle \xrightarrow{-} \langle \texttt{Cast}\ C\ e_1', (h',l')\rangle = q'$$

and analogously

$$\tilde{q} = \langle \texttt{Cast}\ C\ \tilde{e}_1, (\tilde{h},\tilde{l})\rangle \xrightarrow{-} \langle \texttt{Cast}\ C\ \tilde{e}_1', (\tilde{h}',\tilde{l}')\rangle = \tilde{q}',$$

where, by the premise of the rule,

$$\langle e, (h,l)\rangle \xrightarrow{-} \langle e', (h',l')\rangle \quad \text{and} \quad \langle \tilde{e}, (\tilde{h},\tilde{l})\rangle \xrightarrow{-} \langle \tilde{e}', (\tilde{h}',\tilde{l}')\rangle.$$

By definition of $\equiv_f$ (which is by structural isomorphism), we have $e_1 \equiv_f \tilde{e}_1$ and, therefore, $\langle e_1, (h,l)\rangle$ and $\langle \tilde{e}_1, (\tilde{h},\tilde{l})\rangle$ are also f-corresponding. Therefore, by the inductive hypothesis, there exists an $(h',\tilde{h}')$-congruence f' such that $\langle e_1', (h',l')\rangle$ and $\langle \tilde{e}_1', (\tilde{h}',\tilde{l}')\rangle$ are f'-corresponding, where f' is an extension of f. By the definition of $\equiv_{f'}$, we conclude that configurations $\langle \texttt{Cast}\ C\ e_1', (h',l')\rangle$ and $\langle \texttt{Cast}\ C\ \tilde{e}_1', (\tilde{h}',\tilde{l}')\rangle$ are also f'-corresponding.

Rule 36. We have

$$q = \langle \texttt{new}\, C, (h,l)\rangle \xrightarrow{-} \langle addr\ a, (h(a \mapsto\ (C,\ init\text{-}fields\ FDTs)), l)\rangle = q'$$

and

$$\tilde{q} = \langle \texttt{new}\, C, (\tilde{h},\tilde{l})\rangle \xrightarrow{-} \langle addr\ \tilde{a}, (\tilde{h}(\tilde{a} \mapsto\ (C,\ init\text{-}fields\ FDTs)), \tilde{l})\rangle = \tilde{q}',$$

where a and $\tilde{a}$ are fresh references (that is, references unused in h and $\tilde{h}$, respectively).

We extend $(h,\tilde{h})$-congruence f to an $(h',\tilde{h}')$-congruence f' in the following way: (i) $\texttt{dom}(f') = \texttt{dom}(f) \cup \{a\}$; (ii) $\forall r \in \texttt{dom}(f), f(r) = f'(r)$; (iii) $\tilde{a} = f'(a)$. By the definition of $\equiv_{f'}$, we have $a \equiv_{f'} \tilde{a}$. Furthermore, since the rule leaves the stores l and $\tilde{l}$ unchanged, $l \equiv_{f'} \tilde{l}$. Therefore, q' and $\tilde{q}'$ are f'-corresponding.

Rule 40. We have

$$q = \langle V := \texttt{Val}\ v, (h,l)\rangle \xrightarrow{-} \langle unit, (h, l(V \mapsto\ v))\rangle = q'$$

and

$$\tilde{q} = \langle V := \texttt{Val}\ \tilde{v}, (\tilde{h},\tilde{l})\rangle \xrightarrow{-} \langle unit, (\tilde{h}, \tilde{l}(V \mapsto\ \tilde{v}))\rangle = \tilde{q}'$$

with $l \equiv_f \tilde{l}$ by the lemma's hypothesis. Since $q \equiv \tilde{q}$, by the definition of $\equiv_f$ (which is by structural isomorphism), we also have $v \equiv_f \tilde{v}$ and, since $l' = l(V \mapsto v)$ and $\tilde{l}' = \tilde{l}(V \mapsto \tilde{v})$, we also have $l' \equiv_f \tilde{l}'$. Therefore, since the rule leaves unchanged the heaps h and $\tilde{h}$, q' and $\tilde{q}'$ are f'-corresponding where $f' = f$.

Rule 43. We have

$$q = \langle addr\ a.F\{D\} := \texttt{Val}\ v, (h,l)\rangle \xrightarrow{-} \langle unit, (h(a\ \mapsto\ (C, fs((F,D)\ \mapsto\ v))), l)\rangle = q'$$

and

$$\tilde{q} = \langle addr\ \tilde{a}.F\{D\} := \texttt{Val}\ \tilde{v}, (\tilde{h},\tilde{l})\rangle \xrightarrow{-} \langle unit, (\tilde{h}(\tilde{a}\ \mapsto\ (C, fs((F,D)\ \mapsto\ \tilde{v}))), \tilde{l})\rangle = \tilde{q}'.$$

Since $q \equiv \tilde{q}$, by the definition of $\equiv_f$ (which is by structural isomorphism), we have $a \equiv_f \tilde{a}$ and $v \equiv_f \tilde{v}$. The rule changes the heaps $h,\tilde{h}$ in such a way that the fields $F\{D\}$ of the references a,$\tilde{a}$ are updated with the values $v,\tilde{v}$, respectively:

$$h' = h(a\ \mapsto\ (C, fs((F,D)\ \mapsto\ v)))$$

and

$$\tilde{h}' = \tilde{h}(\tilde{a}\ \mapsto\ (C, fs((F,D)\ \mapsto\ \tilde{v}))).$$

Since the two fields are updated with the corresponding values $v,\tilde{v}$, they remain corresponding also after the application of the rule. Therefore, by the definition of $\equiv_f$, the $(h,\tilde{h})$-congruence f is also an $(h',\tilde{h}')$-congruence. Since the rule leaves unchanged the stores l and $\tilde{l}$, $\langle e', (h',l')\rangle = \langle unit, (h',l)\rangle$ and $\langle \tilde{e}', (\tilde{h}',\tilde{l}')\rangle = \langle unit, (\tilde{h}',\tilde{l})\rangle$ are f'-corresponding where $f' = f$.

Rule 44. In this case we have $q = \langle addr\ a.M(map\ \texttt{Val}\ vs), (h,l)\rangle$ and analogously $\tilde{q} = \langle addr\ \tilde{a}.M(map\ \texttt{Val}\ \widetilde{vs}), (\tilde{h},\tilde{l})\rangle$. By the definition of f-corresponding and of $\equiv_f$, we have $a \equiv_f \tilde{a}$ and $vs \equiv_f \widetilde{vs}$. Therefore both a and $\tilde{a}$ point to objects of the same class C defined in E'.

By the premise of the rule we have "$P \vdash\ C\ sees\ M:\ Ts \to T = (pns,\ body)\ in\ D$", which means that method M called for an object of class C (which is the actual class of the object a that we call M for) is defined in class D as $(pns, body)$, where pns are the parameters of the method and $body$ is its body). Now, when configuration q is considered, $P = E' \cdot T$; when $\tilde{q}$ is considered, $P = E' \cdot \tilde{T}$. One can see that in both these cases, the above relation gives the same pns and $body$. This is because, as we have noted, class C is defined in E' and this component does not refer to $T/\tilde{T}$.

Therefore

$$e' = blocks_D(this \cdot pns,\ Class\ D \cdot Ts,\ addr\ a \cdot vs,\ body)$$

and

$$\tilde{e}' = blocks_D(this \cdot pns,\ Class\ D \cdot Ts,\ addr\ \tilde{a} \cdot \widetilde{vs},\ body)$$

respectively. Since $a \equiv_f \tilde{a}$ and $vs \equiv_f \widetilde{vs}$, by the definition of $\equiv_f$ (which is by structural isomorphism), we have $e' \equiv_f \tilde{e}'$. Furthermore, since the rule does not change the state i.e., $(h,l) = (h',l')$ and $(\tilde{h},\tilde{l}) = (\tilde{h}',\tilde{l}')$ respectively, also $l' \equiv_f \tilde{l}'$ holds. Therefore $\langle e', (h',l')\rangle$ and $\langle \tilde{e}', (\tilde{h}',\tilde{l}')\rangle$ are f'-corresponding where $f' = f$.

We consider the remaining rules only quickly, as the reasoning they require is either trivial or similar to the cases discussed above.

- Rules 22-27, 31-35, 55, 57, 75-77 can be proved by following the inductive reasoning (rule induction) for the rule 21, since they also perform just a reduction step in one of their subexpressions.
- Rules 28, 29, 30 cannot be applied to q and $\tilde{q}$ because the function $g(\ell, D)$, which defines the transition's label of these rules, never returns $-$.
- Rules 37, 39, 41, 42, 51, 59, 60, 84-87, 89,90 can be proved by following the reasoning for rule 44 since they also leave the state of the configurations unchanged: $(h,l) = (h',l')$ and $(\tilde{h},\tilde{l}) = (\tilde{h}',\tilde{l}')$ respectively. Therefore, the $(h',\tilde{h}')$-congruence f' is such that $f' = f$.
- Rules 38, 47-50, 52-54, 56, 58, 61-67, 70-74, 78-80, 83 can be proved by following the reasoning for rule 44 because also these rules leave the state of the configurations unchanged (hence, $f' = f$) and, moreover, they are trivial to prove since they do not have any premise (as in case of rule 40).
- Rules 45, 46, 68, 69 cannot be applied to q and $\tilde{q}$ because their transition's label is D.
- Rule 81 can be proved by following the reasoning for rule 44, with the only difference that, since the method $D.M$ is static, the local variable $this$ does not appear as argument of the auxiliary block function.
- Rule 82 can be proved by following the reasoning for rule 36: since also in this case two new array references a and $\tilde{a}$ are created inside their heaps h and $\tilde{h}$ respectively, the $(h,\tilde{h})$-congruence f must be extended in the same way i.e., with a $(h',\tilde{h}')$-congruence f' such that $a \in \texttt{dom}(f')$ and $\tilde{a} = f'(a)$.

– Rule 88 can be proved by following the same reasoning of rule 43: two array references $r = h(a)[n]$ and $\tilde{r} = \tilde{h}(\tilde{a})[n]$ are updated, but also in this case the $(h,\tilde{h})$-congruence f remains unchanged for $(h',\tilde{h}')$ i.e., $f' = f$.

□

E'-configurations. We say that a configuration q is an E'-configuration, if E' has control at q, i.e. $q \xrightarrow{\ell} q'$ for $\ell \in E'$.

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

$$q \xrightarrow{\ell_0} q_1 \xrightarrow{\ell_1} \cdots \xrightarrow{\ell_{n-1}} q_n \xrightarrow{\ell_n} q'$$

where ($\ell_0 \in E'$ and) $\ell_i \notin E'$ for $i \in \{1,\ldots,n\}$, that is $q_1,\ldots,q_n$ are not E'-configurations. (Note that the special case of the above definition is when q' is obtained from q in one step). As we can see, q' is the first E'-configuration after q.

Now one can prove that two E' configurations q and $\tilde{q}$ that are corresponding reduce in one step to configurations which are also corresponding:

Lemma 7. *Let q and $\tilde{q}$ be corresponding E'-configurations. Let $q \to q'$ and $\tilde{q} \to \tilde{q}'$. Then q' and $\tilde{q}'$ are also corresponding.*

Proof. Let $q = \langle e,(h,l)\rangle$ and $\tilde{q} = \langle \tilde{e},(\tilde{h},\tilde{l})\rangle$. Since they are E'-configurations, we have $\langle e,(h,l)\rangle \xrightarrow{D} \langle e',(h',l')\rangle$ and $\langle \tilde{e},(\tilde{h},\tilde{l})\rangle \xrightarrow{\tilde{D}} \langle \tilde{e}',(\tilde{h}',\tilde{l}')\rangle$, respectively, with $D,\tilde{D} \in E'$. By Lemma 5, we then have:

– $e = \mathfrak{C}[e_0]$, where e_0 is a block expression of class D without nested blocks;
– $\tilde{e} = \tilde{\mathfrak{C}}[\tilde{e}_0]$, where $\tilde{e}_0$ is a block expression of class $\tilde{D}$ without nested blocks;
– $\langle e_0,(h,l)\rangle \xrightarrow{D} \langle e'_0,(h',l')\rangle$ and $e' = \mathfrak{C}[e'_0]$;
– $\langle \tilde{e}_0,(\tilde{h},\tilde{l})\rangle \xrightarrow{\tilde{D}} \langle \tilde{e}'_0,(\tilde{h}',\tilde{l}')\rangle$ and $\tilde{e}' = \tilde{\mathfrak{C}}[\tilde{e}'_0]$.

Since $\langle e,(h,l)\rangle$ and $\langle \tilde{e},(\tilde{h},\tilde{l})\rangle$ are corresponding configurations, we have $\mathrm{sub}_{E'}(\mathfrak{C}[e_0]) \equiv_f \mathrm{sub}_{E'}(\tilde{\mathfrak{C}}[\tilde{e}_0])$ for some $(h,\tilde{h})$-congruence f and hence, by the definition of pruning operator, $\mathrm{sub}_{E'}(\mathfrak{C})[\mathrm{sub}_{E'}(e_0)] \equiv_f \mathrm{sub}_{E'}(\tilde{\mathfrak{C}})[\mathrm{sub}_{E'}(\tilde{e}_0)]$. By the definition of $\equiv_f$ (which is by structural isomorphism), we then have:

– $\mathrm{sub}_{E'}(\mathfrak{C}) \equiv_f \mathrm{sub}_{E'}(\tilde{\mathfrak{C}})$ and
– $\mathrm{sub}_{E'}(e_0) \equiv_f \mathrm{sub}_{E'}(\tilde{e}_0)$.

Therefore $\langle e_0,(h,l)\rangle$ and $\langle \tilde{e}_0,(\tilde{h},\tilde{l})\rangle$ are also f-corresponding and $D = \tilde{D}$, because $D,\tilde{D} \in E'$ and, hence, the block expressions e_0 and $\tilde{e}_0$ are preserved by $\mathrm{sub}_{E'}$. Furthermore, since e_0 and $\tilde{e}_0$ contain no block as their proper subexpressions, by the definition of pruning operator, we have $\mathrm{sub}_{E'}(e_0) = e_0$ and $\mathrm{sub}_{E'}(\tilde{e}_0) = \tilde{e}_0$ and therefore $e_0 \equiv_f \tilde{e}_0$.

The relation $\equiv_f$ between expressions implies that e_0 and $\tilde{e}_0$ are syntactically equal, up to reference occurring as their corresponding subexpressions. But, since aplicability of Jinja+ rules does not depend on particular values of references, the same block rule is applied to both $\langle e_0,(h,l)\rangle$ and $\langle \tilde{e}_0,\tilde{h},\tilde{l})\rangle$.

We now prove that $\langle e'_0,(h',l')\rangle$ and $\langle \tilde{e}'_0,(\tilde{h}',\tilde{l}')\rangle$ are f'-corresponding for an $(h',\tilde{h}')$-congruence f' which is an extension of f. We need to distinguish the following cases, depending on the block rule applied to e_0 (and thus to e'_0).

(a) Rule 28, 29: we have $e_0 = \{V : T; e_1\}_D$, where neither $e_1 = \texttt{Val}\ u$ nor $e_1 = \texttt{Throw}\ a$, and $g(\ell, D) = D$.
(b) Rule 45: we have $e_0 = \{V : T;\ \texttt{Val}\ u\}_D$.
(c) Rule 68: we have $e_0 = \{V : T;\ \texttt{Throw}\ a\}_D$.
(d) Rule 30: we have $e_0 = \{V : T; V := \texttt{Val}\ v; e_1\}_D$, where neither $e_1 = \texttt{Val}\ u$ nor $e_1 = \texttt{Throw}\ a$, and $g(\ell, D) = D$.
(e) Rule 46: we have $e_0 = \{V : T;\ V := \texttt{Val}\ v\}_D$.
(f) Rule 69: we have $e_0 = \{V : T; V := \texttt{Val}\ v;\ \texttt{Throw}\ a\}_D$.

Let us consider the case (a) and (b); the case (d) is analogous to case (a), whereas the cases (c), (e) and (f) are analogous to case (b) and, moreover, trivial because the rules do not assume any premise and do not change the state of the configurations.

Case (a): We have $e_0 = \{V : T; e_1\}_D$ and $\tilde{e}_0 = \{V : T; \tilde{e}_1\}_D$. Furthermore, by definition of $\equiv_f$, we have $e_1 \equiv_f \tilde{e}_1$ and hence $\langle e_1, (h, l)\rangle$ and $\langle \tilde{e}_1, (\tilde{h}, \tilde{l})\rangle$ are f-corresponding.

The block rule applied is either 28 or 29. Let us consider the rule 28; reasoning for the other other is analogous. Since e_1 and $\tilde{e}_1$ do not contain any blocks, by Lemma 4

$$\langle e_1, (h, l(V := None)\rangle \xrightarrow{-} \langle e_1', (h', l_1')\rangle \text{ with } l' = l_1'(V := l\, V) \tag{9}$$

and

$$\langle \tilde{e}_1, (\tilde{h}, \tilde{l}(V := None)\rangle \xrightarrow{-} \langle \tilde{e}_1', (\tilde{h}', \tilde{l}_1')\rangle \text{ with } \tilde{l}' = \tilde{l}_1'(V := \tilde{l}\, V). \tag{10}$$

By Lemma 6, $\langle e_1', (h', l_1')\rangle$ and $\langle \tilde{e}_1', (\tilde{h}', \tilde{l}_1')\rangle$ are f'-corresponding for an $(h', \tilde{h}')$-congruence f' which is an extension of f. It implies $\text{sub}_{E'}(e_1') \equiv_{f'} \text{sub}_{E'}(\tilde{e}_1')$ and $l_1' \equiv_{f'} \tilde{l}_1'$. Therefore, by the definition of l' in (9) and $\tilde{l}'$ in (10), we have

$$l' \equiv_{f'} \tilde{l}'. \tag{11}$$

By the definition of $\equiv_{f'}$ (which is by structural isomorphism), we have

$$\{V : T;\ \text{sub}_{E'}(e_1')\}_D \equiv_{f'} \{V : T;\ \text{sub}_{E'}(\tilde{e}_1')\}_D, \tag{12}$$

and, hence, by the definition of the pruning operator, we obtain

$$\text{sub}_{E'}(e_0') = \text{sub}_{E'}(\{V : T; e_1')\}_D) \equiv_{f'} \text{sub}_{E'}(\{V : T; \tilde{e}_1'\}_D) = \text{sub}_{E'}(\tilde{e}_0'). \tag{13}$$

By relations (11) and (13), $\langle e_0', (h', l')\rangle$ and $\langle \tilde{e}_0', (\tilde{h}', \tilde{l}')\rangle$ are also f'-corresponding.

Case (b): We have $e_0 = \{V : T;\ \texttt{Val}\ u\}_D$ and $\tilde{e}_0 = \{V : T;\ \texttt{Val}\ \tilde{u}\}_D$. Furthermore, by the definition of $\equiv_f$, we have $\texttt{Val}\ u \equiv_f \texttt{Val}\ \tilde{u}$. The block rule applied is 45, where $\langle e_0', (h', l')\rangle = \langle \texttt{Val}\ u, (h, l)\rangle$ and $\langle \tilde{e}_0', (\tilde{h}', \tilde{l}')\rangle = \langle \texttt{Val}\ \tilde{u}, (\tilde{h}, \tilde{l})\rangle$, respectively. In particular, since $l \equiv_f \tilde{l}$ (by the lemma's hypothesis) and since $l' = l$ and $\tilde{l}' = \tilde{l}$, we also have $l' \equiv_f \tilde{l}'$.

Therefore $\langle e_0', (h', l')\rangle$ and $\langle \tilde{e}_0', (\tilde{h}', \tilde{l}')\rangle$ are f'-corresponding, where $f' = f$.

We have proved that $\langle e_0', (h', l')\rangle$ and $\langle \tilde{e}_0', (\tilde{h}', \tilde{l}')\rangle$ are f'-corresponding. In particular, it means that $\text{sub}_{E'}(e_0') \equiv_{f'} \text{sub}_{E'}(\tilde{e}_0')$. Since $\text{sub}_{E'}(\mathfrak{C}) \equiv_f \text{sub}_{E'}(\tilde{\mathfrak{C}})$ and f' is an extension of f, by the definition of the pruning operator, we have

$$\text{sub}_{E'}(e') = \text{sub}_{E'}(\mathfrak{C})[\text{sub}_{E'}(e_0')] \equiv_{f'} \text{sub}_{E'}(\tilde{\mathfrak{C}})[\text{sub}_{E'}(\tilde{e}_0')] = \text{sub}_{E'}(\tilde{e}'). \tag{14}$$

By (11) and (14), we conclude that $q' = \langle e', (h', l')\rangle$ and $\tilde{q}' = \langle \tilde{e}', (\tilde{h}', \tilde{l}')\rangle$ are corresponding, which completes the proof. □

Now we are ready to prove the following statement.

Lemma 8. *Let q_0 be the initial configuration of the run of $E' \cdot T$ and $\tilde{q}_0$ be the initial configuration of the run of $E' \cdot \tilde{T}$. Let $q_1, \ldots, q_n$ and $\tilde{q}_1, \ldots, \tilde{q}_n$ be configurations such that $q_0 \mapsto q_1 \mapsto \cdots \mapsto q_n$ and $\tilde{q}_0 \mapsto \tilde{q}_1 \mapsto \cdots \mapsto \tilde{q}_m$ where q_n and $\tilde{q}_m$ are final configuration (i.e. configuration that do not reduce). Then $n = m$ and, for all $i \in \{0, \ldots, n\}$, the configurations q_i and $\tilde{q}_i$ are corresponding.*

Proof. We will prove a more general fact, than the one stated in the lemma, allowing q_0 and $\tilde{q}_0$ to be any corresponding E-states (not necessarily the initial ones). The proof proceeds by induction on the number of E'-blocks—that is block expressions of the form $\{\ldots\}_C$ with $C \in E'$—in q_0 (and $\tilde{q}_0$), where to prove that the statement is true for configurations with a given number of E'-blocks, we assume that it holds true for configurations with bigger numbers of E'-blocks.

If q and $\tilde{q}$ are corresponding and $q \mapsto q'$ where q' is obtained from q in one step, then $\tilde{q} \mapsto \tilde{q}'$ where $\tilde{q}'$ is obtained also in one step. Moreover, by Lemma 7, the configurations q' and $\tilde{q}'$ are corresponding. Therefore, to complete the proof, it is enough to consider the remaining case (i.e., if q' is not obtained from q in one step) and show that calls to the (public) methods of T and $\tilde{T}$ do not brake this property. We consider, on the case by case basis, all calls from E' to methods of $T/\tilde{T}$ (that is to `registerEncryptor` and to `getEncryptor`) made in corresponding states and show that they end in corresponding states as well. Here we present the reasoning only for the former case (the proof for `getEncryptor` proceeds in a similar way).

Method `registerEncryptor` *with arguments* `encryptor, id, pki_domain`*:*
First, we can observe (by inspecting the code of this method in $T/\tilde{T}$, see Appendix D.1 and D.2), these methods do not change the state of E' which formally means, that they preserver condition (a) of the definition of corresponding states. Therefore, it is enough to show that a call to this method also preserves condition (b) of this definition.

This call, in both systems T and $\tilde{T}$ makes three steps:

1. Method `PKI.register` is called with arguments `id`, the PKI domain `pki_domain`, and k, where k is the public key stored in `encryptor`.
   Indeed, in the system T, the control is immediately handed over to `PKISim.register` (with arguments `id`, `pki_domain`, k; see line 155), where method `PKISim.register` is called with the same arguments. In the system $\tilde{T}$, on the other hand, the first things that happens is the call to `register` method of class `RegisterEncSim` (line 64, Appendix D.1) with arguments `id`, `pki_domain`, and k. By the definition of the simulator, this call is directly translated into the corresponding call to method `PKISim.register`.
   As this is the first action in both systems, the configurations of the systems $T/\tilde{T}'$ when this method is called remain corresponding. Hence, by the inductive hypothesis, the state of these systems after the call are corresponding as well. This includes the return value from the method call (as this value has been determined by E' in corresponding states).
   Finally, in both systems, if the return value from the call to `PKISim.register` is true (which is, as we have noticed, the same in both systems), exception `NetwokError`

is thrown. If this is the case, the method call is aborted in corresponding states. Otherwise, the systems enter the next step in corresponding states.

2. It is checked if a key has been already registered for the given `id` and `pki_domain` (line 346 for the system T and line 66 for the system $\tilde{T}$).
   One can notice, again, that this step does not change the state of the system and therefore preserves correspondence of states of $T/\tilde{T}$. Moreover, by condition (b) of the definition of corresponding states, the result of this step is the same in both systems. Therefore either both T and $\tilde{T}$ throw `PKIError` in corresponding states, or both T and $\tilde{T}$ enter the next step in corresponding states.
3. A public key/encryptor is registered under `id` and `pki_domain`.
   This is done in line 347 for the system T and in line 68 for the system $\tilde{T}$. One can see, by inspecting the code of the invoked methods, that the only part of the state that is changed are the collections considered in condition (b) of the definition of corresponding states. So, condition (a) of this definition is preserved by this step. Moreover, the changes made to the considered collections are such that condition (b) is also preserved after this step. Hence, the states of T and $\tilde{T}$ after this step are still corresponding.

Now we can complete the proof of Lemma 2. By the above lemma, the final configurations of $(E \cdot S) = (E' \cdot T)$ and $(E \cdot \tilde{S}) = (E' \cdot \tilde{T})$ (that is, respectively, q_n and $\tilde{q}_m$, as defined in the lemma) are corresponding, which means, in particular, that the state of E', which includes the variable `result` in those configurations is the same. Therefore the environment outputs the same result in both cases. □

B.2 Proof of Lemma 3

In this section, we provide the proof of the realization result for public-key encryption:

$$\mathsf{Real\text{-}PKIEnc} \leq^{I_{PKIEnc}} \mathsf{Ideal\text{-}PKIEnc}. \tag{15}$$

In our previous paper [24] we considered the case without corruption. In this paper, we consider an extended case with (static) corruption: in our Jinja+ implementation, we model corruption by allowing the direct creation of `Encryptor` objects with an arbitrary public key provided by the adversary.

We structure the proof in the following way: first we discuss the functionalities without corruption by referencing to a result obtained in [24] and then, based on this result, we consider the case with corruption and prove Lemma 3.

The Functionalities without Corruption. The real functionality of public key encryption without corruption, as considered in [24] coincides with the real functionality with corruption we consider in this paper. The ideal functionality for public key encryption without corruption, as considered in [24], is, however, different (see Appendix D.10). We will denote it by $\mathsf{Ideal\text{-}PKEnc}^{-}$.

Similarly, the interface they implement (again, as considered in [24]) will be denoted by I^{-}_{PKEnc}. For completeness, we recall this interface (note that the difference to I_{PKEnc} is the lack of the constructor of class `Encryptor`):

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

We have the following result proven in [24]:

Lemma 9. Real-PKEnc $\leq^{I^{-}_{PKIEnc}}$ Ideal-PKEnc^{-}

The Functionalities with Corruption. Now, using the result just discussed, we prove (15). That is, we show that there exists a probabilistic polynomially bounded simulator S such that for each polynomially bounded I_{PKEnc}-environment E we have (Section 2):

$$E \cdot \text{Real-PKEnc} \equiv_{\text{comp}} E \cdot S \cdot \text{Ideal-PKEnc} \tag{16}$$

In order to reduce this proof to the case without corruption, we take an arbitrary I_{PKEnc}-environment E and construct a new I^{-}_{PKEnc}-environment E^{-} out of it. This environment E^{-} consists of the following parts:

1. A copy of the code of the class `Encryptor` from the real functionality renamed `EncryptorCorr`. (Note that the code of class `Encryptor` in the real functionality and the ideal is identical).
2. A new class `EncryptorWrapper` which is meant to wrap either an object of class `Encryptor` of the interface I^{-}_{PKEnc} (objects of this class are returned by `PKI.Decryptor.getEncryptor()`), or an object of class `EncryptorCorr`, as introduced above.

```
public class EncryptorWrapper {
  Encryptor enc;
  EncryptorCorr encCorr;

  public EncryptorWrapper(Encryptor enc) {
    this.enc=enc;
    this.encCor=null;
  }
  public EncryptorWrapper(EncryptorCorr encCorr) {
    this.encCorr=encCorr;
    this.enc=null;
  }
  public byte[] encrypt(byte[] message) {
    if(enc!=null)
      return enc.encrypt(message);
    else
      return encCorr.encrypt(message);
```

```
18    }
19    public byte[] getPublicKey(){
20      if(enc!=null)
21        return enc.getPublicKey();
22      else
23        return encCorr.getPublicKey();
24    }
25  }
```

3. A copy of the code of E modified in the following way:
   (a) every expression where an encryptor is obtained by a decryptor i.e.,
   `decryptor.getEncryptor()`
   is replaced by
   `new EncryptorWrapper(decryptor.getEncryptor())`
   (b) every expression where a corrupted encryptor is directly created i.e.,
   `EncryptorCorr(pubk)`
   is replaced by
   `new EncryptorWrapper(new EncryptorCorr(pubk));`

The reason for using the wrapper class is to make it possible to treat objects of two, formally unrelated classes (the encryptor class provided by the environment and the encryptor class provided by the functionality) in a uniform way.

Using this construction, we can state the two following lemmas.

Lemma 10. $E \cdot \mathsf{Real\text{-}PKEnc} \equiv_{\mathsf{comp}} E^{-} \cdot \mathsf{Real\text{-}PKEnc}$

Proof (sketch). The proof is quite straightforward and it it follows by the construction of E^{-}. The class `EncryptorCorr` contains a copy of the code of class `Encryptor` of Real-PKEnc. Furthermore, the wrapper and the modified version of E perform the same actions (up to additional relaying steps of the wrapper class).

The presented above reasoning can be strictly formalized, as it has been in the proof of Lemma 2. The difference to the proof of Lemma 2 is that now we cannot prove perfect indistinguishability of the considered system, but the following property (which is still stronger than the one postulated in the lemma): the considered systems behave in exactly the same way from the point of view of an unbounded (but possibly probabilistic) adversary, for the same sequence of random coins.

In a very similar way we can prove the following result.

Lemma 11. $E \cdot S \cdot \mathsf{Ideal\text{-}PKEnc} \equiv_{\mathsf{comp}} E^{-} \cdot S \cdot \mathsf{Ideal\text{-}PKEnc}^{-}$

Now we are ready to complete the proof of Lemma 3. From Lemma 9, we know that $\mathsf{Real\text{-}PKEnc} \leq^{I^{-}_{PKEnc}} \mathsf{Ideal\text{-}PKEnc}^{-}$ i.e., exists a probabilistic polynomially bounded simulator S such that for each polynomially bounded I^{-}_{PKEnc}-environment E^{-} we have (Section 2):

$$E^{-} \cdot \mathsf{Real\text{-}PKEnc} \equiv_{\mathsf{comp}} E^{-} \cdot S \cdot \mathsf{Ideal\text{-}PKEnc}^{-} \tag{17}$$

Therefore, we obtain:

$$\begin{aligned} E \cdot \mathsf{Real\text{-}PKEnc} &\overset{\text{Lemma 10}}{\equiv_{\mathsf{comp}}} E^{-} \cdot \mathsf{Real\text{-}PKEnc} \\ &\overset{(17)}{\equiv_{\mathsf{comp}}} E^{-} \cdot S \cdot \mathsf{Ideal\text{-}PKEnc}^{-} \\ &\overset{\text{Lemma 11}}{\equiv_{\mathsf{comp}}} E \cdot S \cdot \mathsf{Ideal\text{-}PKEnc} \end{aligned} \tag{18}$$

C Proof of Theorem 5

The proof of Theorem 5 is, as it is in the case of the realization result for PKIEnc, modular and uses such properties of the realization relation as the composition theorem, reflexivity, and transitivity.

We begin with analyzing the structure of the ideal and real functionalities we consider. The ideal functionality consists of the following components:

- Ideal-Sig — the (ideal) implementation of digital signatures, i.e. classes `Verifier` and `Signer`, as described in Section 4.2. Let $I_{\mathsf{CryptoLibSig}}$ denote the public interface of this component.
- Ideal-SigPKI — the (ideal) implementation of verifier registration, that is of the class `RegisterSig`. Let I_{SigPKI} denotes the public interface of this component.

Similarly, the real functionality for PKISig consists of the following components:

- Real-Sig — the (real) implementation of classes `Verifier` and `Signer`, as sketched in Section 4.3. Note that the public interface of this component is $I_{\mathsf{CryptoLibSig}}$, as in the case of Ideal-Sig.
- Real-SigPKI — the (real) implementation of verifier registration, that is `RegisterSig`. This implementation uses the next component, Real-PKI, and only wraps/unwraps verification keys into/from verifiers. Note that the public interface of this component is I_{SigPKI}.
- Real-PKI — The real implementation of the functionality for the public key infrastructure (see Section 3.3).

Now, proving Theorem 5 can be reduced (in an analogous way as in the proof of Theorem 4) to proving the following two facts:

Lemma 12. $\mathsf{Real\text{-}Sig} \cdot \mathsf{Real\text{-}SigPKI} \cdot \mathsf{Ideal\text{-}PKI} \leq^{I_{PKISig}} \mathsf{Real\text{-}Sig} \cdot \mathsf{Ideal\text{-}SigPKI}$.

The proof of this lemma is very similar to the proof of Lemma 2 given in Appendix B.1.

Lemma 13. $\mathsf{Real\text{-}Sig} \leq^{I_{PKISig}} \mathsf{Ideal\text{-}Sig}$.

The rest of this section is devoted to proving this lemma. We organize this proof in a similar way to the proof of Lemma 3. First, we discuss the case without corruption, where the adversary (the environment) cannot create verifiers with arbitrary verification keys. Then we extend the proof to the case with (static) corruption, where the adversary can directly create verifiers with an arbitrary verification keys.

C.1 The Functionality without Corruption

As in the proof of Lemma 3, we denote ideal functionality without corruption as Ideal-Sig$^-$ (see Appendix D.10 for the code). Similarly, the interface the real and ideal functionalities for digital signatures without corruption implement is denoted by $I^-_{\text{CryptoLibSig}}$.

```
public final class Signer {
  public Signer();
  public byte[] sign(byte[] message);
  public Verifier getVerifier();
}
public class Verifier {
  public boolean verify(byte[] signature, byte[] message);
  public byte[] getVerifKey();
}
```

Note that the only difference to the interface $I_{\text{CryptoLibSig}}$ is the lack of the constructor of the class `Verifier`.

To prove the following theorem we use here a proof technique similar to those used in [24]: we reduce the problem to the corresponding problem stated in the Turing Machine representation in order to use a result from [21, 26].

Lemma 14. Real-Sig $\leq^{I^-_{PKISig}}$ Ideal-Sig$^-$

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

1. First, we assume that we have a correct implementation of an UF-CMA-secure (existential unforgeability under adaptive chosen-message attacks) digital signatures 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 signature scheme is such that the length of a signature is the (polynomially computable) function of the length of the signed message 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 an ideal functionality $\mathscr{F}$ and a real functionality $\mathscr{R}$ for digital signatures in the Turing machine model following [21, 26].

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-Signer*: The functionality is supposed to return a verification key *vk*.
2. *Initialization-Verifier* The functionality responds with the message "*comleted*".

3. *Signature-Generation(m)*: The functionality is supposed to sign m using the stored signing key and return the signature σ.
4. *Signature-Verification(vk, m, σ)*: The functionality is supposed to verify that σ is a valid signature for the message m under the verification key vk.

Both the real and the ideal functionalities, on initialization, obtain a corruption bit. As already explained at the beginning of this section, because for now we handle the case without corruption i.e., 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-Signer* or *Initialization-Verifier*), generates a fresh verification/signing key pair and returns the verification key. Then, it uses the signing key to sign messages, and the key vk provided in the verification request to verify messages.

The ideal functionality, on creation, asks the environment (the simulator) for verification and signing algorithms as well as a verification (public) and a signing (private) key. On signing requests, it (similarly to the considered Jinja+ functionality) computes a signature for the message provided using the given signing algorithm and private key. Then, by using the recorded verification algorithm and public key, it checks whether the signature verifies or not. If this check fails, it returns an error message. Otherwise, it records the message provided (to prevent forgery) and returns the signature. On verification, if the key vk is the same as the verification key stored in the functionality, it verifies the signature σ for m using the provided vk and checks that m has been stored as signed; see [21, 26] for details.

We want to prove that Real-Sig realizes Ideal-Sig$^-$ w.r.t. I^-_{PKISig}. In this proof we will use a result from [21, 26] that $\mathscr{R}$ realizes $\mathscr{F}$. Let $\mathscr{S}$ be the simulator used in the realization proof in [21, 26]. The simulator for Ideal-Sig$^-$ we will use in the proof is $S = $ UF-CMA, as described above.

Let E be a bounded-environment with $I^-_{PKISig} \vdash E$.

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 Real-Sig or Ideal-Sig$^-$). 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 Signer or Verifier. For each reference to an internal object, a representation of this object is maintained by M_E. For references to external objects this is not the case (some additional information, however, is stored along with these references; see below). A method call for an internal reference is

modelled internally by 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 `Signer`:** M_E creates a new instance of *Signer* (Turing Machine) by sending the *Initialization-Signer* request with a fresh identifier *id*. This identifier will be used as the reference to this object. M_E waits then for a response containing a verification key. This key is stored together with *id*.
2. **`Signer.getVerifier` for an object represented by** *id***:** M_E creates a new instance of *Verifier* (TM) by sending the *Initialization-Verifier* 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. **`Signer.sign` for an object represented by** *id* **and array** m**:** M_E sends *Signature-Generation* request to machine *id* with the data stored under m, 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. **`Verifier.verify` for an object represented by** id' **and two arrays** m **and** σ**:** M_E retrieves the verification key associated with id' and uses it in the request *Signature-Verification* along with id' and the data stored under both m (the message) and σ (the signature). A response is one bit. M_E retrieves the response.
5. **`Verifier.getVerifKey` for an object represented by** id'**:** M_E retrieves the verified key associated with the Verifier (without any external call).

Representing runs. Let T be either the system Real-Sig or the system $(S \cdot \text{Ideal-Sig}^{-})$. Let u be a random input (a sequence of bits) and η be a security parameter. 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_i 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 $r=f(\tilde{r})$ (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 field 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 15. *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 Real-Sig and $\mathscr{S}|\mathscr{F}$ produces the same data as $S \cdot \text{Ideal-Sig}^{-}$. 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 $E \cdot \text{Real-Sig}$ outputs true with the same probability the system $M_E|\mathscr{R}$ outputs 1 and the system $E \cdot S \cdot \text{Ideal-Sig}$ 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 $E \cdot \text{Real-Sig}$ and by $E \cdot S \cdot \text{Ideal-Sig}$ is the same up to some negligible value.

C.2 Proof of Lemma 13

As in the realization proof for public-key encryption, we take an $I_{\text{CryptoLibSig}}$-environment E and we construct a $I^{-}_{\text{CryptoLibSig}}$-environment E' which consists of (1) a copy of the class `Verifier` (renamed as `VerifierCorr`), (2) a wrapper class `VerifierWrapper` providing unified access to corrupted and uncorrupted verifiers, and (3) an appropriately aligned copy of E (as in Appendix B.2). Using this construction, we obtain results analogous to Lemmas 10 and 11:

Lemma 16. $E \cdot \mathsf{Real\text{-}Sig} \equiv_{\mathsf{comp}} E' \cdot \mathsf{Real\text{-}Sig}$

Lemma 17. $E \cdot S \cdot \mathsf{Ideal\text{-}Sig} \equiv_{\mathsf{comp}} E' \cdot S \cdot \mathsf{Ideal\text{-}Sig}^{-}$

From Lemma 14 we know that $\mathsf{Real\text{-}Sig} \leq^{I^{-}_{PKISig}} \mathsf{Ideal\text{-}Sig}^{-}$, i.e. there exists a probabilistic polynomially bounded simulator S such that for each polynomially bounded I^{-}_{PKISig}-environment E' we have (Section 2):

$$E' \cdot \mathsf{Real\text{-}Sig} \equiv_{\mathsf{comp}} E' \cdot S \cdot \mathsf{Ideal\text{-}Sig}^{-} \tag{19}$$

Therefore we obtain

$$\begin{aligned} E \cdot \mathsf{Real\text{-}Sig} &\overset{\text{Lemma 16}}{\equiv_{\mathsf{comp}}} E' \cdot \mathsf{Real\text{-}Sig} \\ &\overset{(19)}{\equiv_{\mathsf{comp}}} E' \cdot S \cdot \mathsf{Ideal\text{-}Sig}^{-} \\ &\overset{\text{Lemma 17}}{\equiv_{\mathsf{comp}}} E \cdot S \cdot \mathsf{Ideal\text{-}Sig} \end{aligned} \tag{20}$$

which completes the proof of Lemma 13.

D Code of the Functionalities

We present here the most important parts of the code of the proposed functionalities. This code should be sufficient to understand the functionalities and follow the proofs; see [23] for the full code.

D.1 Ideal Functionality for PKIEnc

```
public class Encryptor {
  protected byte[] publicKey;
  public Encryptor(byte[] publicKey) {
    this.publicKey = publicKey;
  }
  public byte[] encrypt(byte[] message) {
    return copyOf(CryptoLib.pke_encrypt(copyOf(message),
                                        copyOf(publicKey)));
  }
  public byte[] getPublicKey() {
    return copyOf(publicKey);
  }
  protected Encryptor copy() {
    return new Encryptor(publicKey);
  }
}
```

—

```
public final class UncorruptedEncryptor extends Encryptor {
  private Decryptor.EncryptionLog log;

  UncorruptedEncryptor(byte[] publicKey, Decryptor.EncryptionLog log) {
    super(publicKey);
    this.log = log;
  }
  public byte[] encrypt(byte[] message) {
    byte[] randomCipher = null;
    while( randomCipher==null || log.containsCiphertext(randomCipher) ) {
      randomCipher = copyOf(CryptoLib.pke_encrypt(getZeroMessage(message.length),
                            copyOf(publicKey)));
    }
    log.add(copyOf(message), randomCipher);
    return copyOf(randomCipher);
  }
  protected Encryptor copy() {
    return new UncorruptedEncryptor(publicKey, log);
  }
}
```

—

```
public class Decryptor {
  private byte[] publicKey;
  private byte[] privateKey;
  private EncryptionLog log;

  public Decryptor() {
    KeyPair keypair = CryptoLib.pke_generateKeyPair();
    this.privateKey = copyOf(keypair.privateKey);
    this.publicKey = copyOf(keypair.publicKey);
    this.log = new EncryptionLog();
  }
  public byte[] decrypt(byte[] message) {
    byte[] messageCopy = copyOf(message);
    if (!log.containsCiphertext(messageCopy)) {
      return copyOf(CryptoLib.pke_decrypt(copyOf(privateKey), messageCopy));
    } else {
      return copyOf(log.lookup(messageCopy));
    }
  }
  public Encryptor getEncryptor() {
    return new UncorruptedEncryptor(publicKey, log);
  }
}
```

—

```
public class RegisterEnc {
  public static void registerEncryptor(Encryptor encryptor, int id,
                    byte[] pki_domain) throws PKIError, NetworkError
```

```
{
  if( RegisterEncSim.register(id, pki_domain, encryptor.getPublicKey()) )
    throw new NetworkError();
  if( registeredAgents.fetch(id, pki_domain) != null )
    throw new PKIError();
  registeredAgents.add(id, pki_domain, encryptor);
}
public static Encryptor getEncryptor(int id, byte[] pki_domain)
                        throws PKIError, NetworkError
{
  if( RegisterEncSim.getEncryptor(id, pki_domain) )
      throw new NetworkError();
  Encryptor enc = registeredAgents.fetch(id, pki_domain);
  if (enc == null)
    throw new PKIError();
  return enc.copy();
}

public static class PKIError extends Exception { }

/// IMPLEMENTATION
private static class RegisteredAgents {
  private static class EncryptorList {
    final int id;
    byte[] domain;
    Encryptor encryptor;
    EncryptorList next;
    EncryptorList(int id, byte[] domain, Encryptor encryptor,
                  EncryptorList next) {
      this.id = id;
      this.domain = domain;
      this.encryptor= encryptor;
      this.next = next;
    }
  }
  private EncryptorList first = null;

  public void add(int id, byte[] domain, Encryptor encr) {
    first = new EncryptorList(id, domain, encr, first);
  }

  Encryptor fetch(int ID, byte[] domain) {
    for( EncryptorList node = first;  node != null; node = node.next ) {
      if( ID == node.id && MessageTools.equal(domain, node.domain) )
        return node.encryptor;
    }
    return null;
  }
}

```

```
  private static RegisteredAgents registeredAgents = new RegisteredAgents();
}
```

D.2 Real Functionality for PKIEnc

```
public class Encryptor {
  private byte[] publicKey;

  public Encryptor(byte[] publicKey) {
    this.publicKey = publicKey;
  }
  public byte[] encrypt(byte[] message) {
    return copyOf(CryptoLib.pke_encrypt(copyOf(message),
                                        copyOf(publicKey)));
  }
  public byte[] getPublicKey() {
    return copyOf(publicKey);
  }
}
```

—

```
public class Decryptor {
  byte[] publicKey;
  byte[] privateKey;

  public Decryptor() {
    KeyPair keypair = CryptoLib.pke_generateKeyPair();
    this.privateKey = copyOf(keypair.privateKey);
    this.publicKey = copyOf(keypair.publicKey);
  }
  Decryptor(byte[] pubk, byte[] prvkey) {
    this.publicKey = pubk;
    this.privateKey = prvkey;
  }
  public byte[] decrypt(byte[] message) {
    return copyOf(CryptoLib.pke_decrypt(copyOf(message),
                                        copyOf(privateKey)));
  }
  public Encryptor getEncryptor() {
    return new Encryptor(copyOf(publicKey));
  }
}
```

—

```
public class RegisterEnc {
  public static void registerEncryptor(Encryptor encryptor, int id,
                        byte[] pki_domain) throws PKIError, NetworkError
  {
    try {
```

```
      PKI.register(id, pki_domain, encryptor.getPublicKey());
    } catch (PKI.Error e) {
      throw new PKIError();
    }
  }
  public static Encryptor getEncryptor(int id, byte[] pki_domain)
                            throws PKIError, NetworkError
  {
    try {
      byte[] key = PKI.getKey(id, pki_domain);
      return new Encryptor(key);
    } catch (PKI.Error e) {
      throw new PKIError();
    }
  }

  public static class PKIError extends Exception { }
}
```

D.3 Ideal Functionality for PKISig

```
public class Verifier {
  protected byte[] verifKey;

  public Verifier(byte[] verifKey) {
    this.verifKey = verifKey;
  }
  public boolean verify(byte[] signature, byte[] message) {
    return CryptoLib.verify(message, signature, verifKey);
  }
  public byte[] getVerifKey() {
    return copyOf(verifKey);
  }
  protected Verifier copy() {
    return new Verifier(verifKey);
  }
}
```

```
public final class UncorruptedVerifier extends Verifier {
  private Signer.Log log;

  UncorruptedVerifier(byte[] verifKey, Signer.Log log) {
    super(verifKey);
    this.log = log;
  }
  public boolean verify(byte[] signature, byte[] message) {
    return CryptoLib.verify(message, signature, verifKey)
            && log.contains(message);
  }
```

```
201   protected Verifier copy() {
202     return new UncorruptedVerifier(verifKey, log);
203   }
204 }

205 final public class Signer {
206   private byte[] verifKey;
207   private byte[] signKey;
208   private Log log;
209
210   public Signer() {
211     KeyPair keypair = CryptoLib.generateSignatureKeyPair();
212     this.signKey = copyOf(keypair.privateKey);
213     this.verifKey = copyOf(keypair.publicKey);
214     this.log = new Log();
215   }
216   public byte[] sign(byte[] message) {
217     byte[] signature = CryptoLib.sign(copyOf(message), copyOf(signKey));
218     if (signature == null) return null;
219     if( !CryptoLib.verify(copyOf(message), copyOf(signature), copyOf(verifKey)) )
220       return null;
221     log.add(copyOf(message));
222     return copyOf(copyOf(signature));
223   }
224   public Verifier getVerifier() {
225     return new UncorruptedVerifier(verifKey, log);
226   }
227 }

228 public class RegisterSig {
229
230   public static void registerVerifier(Verifier verifier, int id,
231                       byte[] pki_domain) throws PKIError, NetworkError
232   {
233     if( RegisterSigSim.register(id, pki_domain, verifier.getVerifKey()) )
234       throw new NetworkError();
235     if( registeredAgents.fetch(id, pki_domain) != null )
236       throw new PKIError();
237     registeredAgents.add(id, pki_domain, verifier);
238   }
239   public static Verifier getVerifier(int id, byte[] pki_domain)
240                           throws PKIError, NetworkError
241   {
242     if( RegisterSigSim.getVerifier(id, pki_domain) ) throw new NetworkError();
243     Verifier verif = registeredAgents.fetch(id, pki_domain);
244     if (verif == null)
245       throw new PKIError();
246     return verif.copy();
247   }
248
```

```
  public static class PKIError extends Exception { }

  /// IMPLEMENTATION ///
  private static class RegisteredAgents {
    private static class VerifierList {
      final int id;
      byte[] domain;
      Verifier verifier;
      VerifierList  next;
      VerifierList(int id, byte[] domain,  Verifier verifier,
                   VerifierList next)
      {
        this.id = id;
        this.domain = domain;
        this.verifier = verifier;
        this.next = next;
      }
    }

    private VerifierList first = null;

    public void add(int id, byte[] domain, Verifier verif) {
      first = new VerifierList(id, domain, verif, first);
    }
    Verifier fetch(int ID, byte[] domain) {
      for( VerifierList node = first;  node != null; node = node.next ) {
        if( ID == node.id && MessageTools.equal(domain, node.domain) )
          return node.verifier;
      }
      return null;
    }
  }

  private static RegisteredAgents registeredAgents = new RegisteredAgents();
}
```

D.4 Real Functionality for PKISig

```
public class Verifier {
  private byte[] verifKey;

  public Verifier(byte[] verifKey) {
    this.verifKey = verifKey;
  }
  public boolean verify(byte[] signature, byte[] message) {
    return CryptoLib.verify(copyOf(message), copyOf(signature), copyOf(verifKey));
  }
  public byte[] getVerifKey() {
    return copyOf(verifKey);
```

```
  }
}
```

—

```
public class Signer {
  byte[] verifKey;
  byte[] signKey;

  public Signer() {
    KeyPair keypair = CryptoLib.generateSignatureKeyPair();
    this.signKey = copyOf(keypair.privateKey);
    this.verifKey = copyOf(keypair.publicKey);
  }
  Signer(byte[] verifKey, byte[] signKey ) {
    this.verifKey = verifKey;
    this.signKey = signKey;
  }
  public byte[] sign(byte[] message) {
    byte[] signature = CryptoLib.sign(copyOf(message), copyOf(signKey));
    return copyOf(signature);
  }
  public Verifier getVerifier() {
    return new Verifier(verifKey);
  }
}
```

—

```
public class RegisterSig {
  public static void registerVerifier(Verifier verifier, int id,
                     byte[] pki_domain) throws PKIError, NetworkError
  {
    try {
      PKI.register(id, pki_domain, verifier.getVerifKey());
    } catch (PKI.Error e) {
      throw new PKIError();
    }
  }
  public static Verifier getVerifier(int id, byte[] pki_domain)
                                          throws PKIError, NetworkError
  {
    try {
      byte[] key = PKI.getKey(id, pki_domain);
      return new Verifier(key);
    } catch (PKI.Error e) {
      throw new PKIError();
    }
  }

  public static class PKIError extends Exception { }
}
```

D.5 Ideal Functionality for PKI

```
public class IdealPKI {
  static void register(int id, byte[] domain, byte[] key)
        throws PKIError, NetworkError
  {
    if (PKISim.register(id, domain, key)) throw new NetworkError();
    if (registered(id, domain)) throw new PKIError();
    entries.add(id, domain, key);
  }
  static byte[] getKey(int id, byte[] domain) throws PKIError, NetworkError {
    if (PKISim.getKey(id, domain)) throw new NetworkError();
    byte[] key = entries.getKey(id, domain);
    if (key == null) throw new PKIError();
    return key;
  }
  static private boolean registered(int id, byte[] domain) {
    return entries.getKey(id, domain) != null;
  }

  /// IMPLEMENTATION ///
  private static class Entry {
    final int id;
    byte[] domain;
    byte[] key;

    Entry(int id, byte[] domain, byte[] key) {
      this.id = id;
      this.domain = domain;
      this.key = key;
    }
  }

  private static class EntryList {
    private static class Node {
      Entry entry;
      Node next;
      Node(Entry entry, Node next) {
        this.entry = entry;
        this.next = next;
      }
    }

    private Node first = null;

    void add(int id, byte[] domain, byte[] key) {
      first = new Node(new Entry(id,domain,key), first);
    }
    byte[] getKey(int id, byte[] domain) {
```

```
      for( Node node=first;  node!=null;  node = node.next ) {
        if ( node.entry.id==id &&  MessageTools.equal(node.entry.domain, domain) ) {
          return node.entry.key;
        }
      }
      return null;
    }
  }

  static private EntryList entries = new EntryList();
}
```

D.6 Ideal Functionality for Private Symmetric Encryption

```
public class SymEnc {
  private byte[] key;
  private EncryptionLog log;

  public SymEnc() {
    key = CryptoLib.symkey_generateKey();
  }
  public byte[] encrypt(byte[] plaintext) {
    byte[] randomCipher = null;
    while( randomCipher==null || log.containsCiphertext(randomCipher) ) {
      randomCipher = copyOf(CryptoLib.symkey_encrypt(copyOf(key),
                            getZeroMessage(plaintext.length)));
    }
    log.add(copyOf(plaintext), randomCipher);
    return copyOf(randomCipher);
  }
  public byte[] decrypt(byte[] ciphertext) {
    if (!log.containsCiphertext(ciphertext)) {
      return copyOf( CryptoLib.symkey_decrypt(copyOf(key), copyOf(ciphertext)) );
    } else {
      return copyOf( log.lookup(ciphertext) );
    }
  }
}
```

D.7 Real Functionality for Private Symmetric Encryption

```
public class SymEnc {
  private byte[] key;

  public SymEnc() {
    key = CryptoLib.symkey_generateKey();
  }
  public byte[] encrypt(byte[] plaintext) {
```

```
    return CryptoLib.symkey_encrypt(copyOf(key), copyOf(plaintext));
  }
  public byte[] decrypt(byte[] ciphertext) {
    return CryptoLib.symkey_decrypt(copyOf(key), copyOf(ciphertext));
  }
}
```

D.8 Ideal Functionality for Nonce Generation

```
public class NonceGen {
  public NonceGen() {
  }
  public byte[] newNonce() {
    byte[] nonce = null;
    // keep asking for a nonce until we get a fresh value
    while( nonce==null || log.contains(nonce) ) {
      nonce = CryptoLib.newNonce();
    }
    log.add(nonce); // log the nonce
    return nonce;
  }
}
```

D.9 Real Functionality for Nonce Generation

```
public class NonceGen {
  public NonceGen() {
  }
  public byte[] newNonce() {
    return CryptoLib.newNonce();
  }
}
```

D.10 Ideal Functionality for Public-key Encryption without Corruption [24]

```
public final class Encryptor {
  private Decryptor.EncryptionLog log;

  Encryptor(byte[] publicKey, Decryptor.EncryptionLog log) {
    super(publicKey);
    this.log = log;
  }

  public byte[] encrypt(byte[] message) {
    byte[] randomCipher = null;
    while( randomCipher==null || log.containsCiphertext(randomCipher) ) {
      randomCipher = copyOf(CryptoLib.pke_encrypt(getZeroMessage(message.length),
```

```
                                copyOf(publicKey)));
    }
    log.add(copyOf(message), randomCipher);
    return copyOf(randomCipher);
  }

  protected Encryptor copy() {
    return new Encryptor(publicKey, log);
  }
}
```

The class `Decryptor` is as in the functionality with corruption (Appendix D.1)

D.11 Ideal Functionality for Digital Signatures without Corruption

```
public final class Verifier {
  private Signer.Log log;

  Verifier(byte[] verifKey, Signer.Log log) {
    super(verifKey);
    this.log = log;
  }
  public boolean verify(byte[] signature, byte[] message) {
    // verify both that the signature is correct
    // and that the message has been logged as signed
    return CryptoLib.verify(message, signature, verifKey)
           && log.contains(message);
  }
  protected Verifier copy() {
    return new Verifier(verifKey, log);
  }
}
```

The class `Signer` is as in the functionality with corruption (Appendix D.3)

E Jinja+

E.1 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 [19], 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 [28].

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 [28], 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.

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 [19] (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).

E.2 Rules of Jinja

There are two points where our presentation rules diverge from the ones of [19]. First, as we assume *unbounded memory*, we do not have rules which throw `OutOfMemoryError` (and we assume that $(\mathit{new\text{-}Addr}\ h)$ 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 (28)–(30)).

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 2) 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 3) 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 auxiliary function *blocks*:

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

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

Exceptional reduction and *exception propagation* rules (Figure 4 and 5) 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.

E.3 Rules of Jinja+

In this section we present additional rules of Jinja+. Theres rules concern static method invocation and arrays. The rules are given in Figure 7 and 6.

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle \texttt{Cast}\ C\ e,s\rangle \xrightarrow{\ell} \langle \texttt{Cast}\ C\ e',s'\rangle} \tag{21}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle V := e,s\rangle \xrightarrow{\ell} \langle V := e',s'\rangle} \tag{22}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle e.F\{D\},s\rangle \xrightarrow{\ell} \langle e'.F\{D\},s'\rangle} \tag{23}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle e.F\{D\} := e_2,s\rangle \xrightarrow{\ell} \langle e'.F\{D\} := e_2,s'\rangle} \tag{24}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle \texttt{Val}\ v.F\{D\} := e,s\rangle \xrightarrow{\ell} \langle \texttt{Val}\ v.F\{D\} := e',s'\rangle} \tag{25}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle e \ll bop \gg e_2,s\rangle \xrightarrow{\ell} \langle e' \ll bop \gg e_2,s'\rangle} \tag{26}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle \texttt{Val}\ v_1 \ll bop \gg e,s\rangle \xrightarrow{\ell} \langle \texttt{Val}\ v_1 \ll bop \gg e',s'\rangle} \tag{27}$$

$$\frac{P \vdash \langle e,(h,l(V := None))\rangle \xrightarrow{\ell} \langle e',(h',l')\rangle \quad l'\ V = None \quad \neg\ assigned\ V\ e}{P \vdash \langle \{V : T; e\}_D,(h,l)\rangle \xrightarrow{g(\ell,D)} \langle \{V : T; e'\}_D,(h',l'(V := l\ V))\rangle} \tag{28}$$

$$\frac{P \vdash \langle e,(h,l(V := None))\rangle \xrightarrow{\ell} \langle e',(h',l')\rangle \quad l'\ V = v \quad \neg\ assigned\ V\ e}{P \vdash \langle \{V : T; e\}_D,(h,l)\rangle \xrightarrow{g(\ell,D)} \langle \{V : T;\ V := \texttt{Val}\ v; e'\}_D,(h',l'(V := l\ V))\rangle} \tag{29}$$

$$\frac{P \vdash \langle e,(h,l(V := v))\rangle \xrightarrow{\ell} \langle e',(h',l')\rangle \quad l'\ V = v'}{P \vdash \langle \{V : T; V := \texttt{Val}\ v; e\}_D,(h,l)\rangle \xrightarrow{g(\ell,D)} \langle \{V : T;\ V := \texttt{Val}\ v'; e'\}_D,(h',l'(V := l\ V))\rangle} \tag{30}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle e.M(es),s\rangle \xrightarrow{\ell} \langle e'.M(es),s'\rangle} \tag{31}$$

$$\frac{P \vdash \langle es,s\rangle \ [\xrightarrow{\ell}]\ \langle es',s'\rangle}{P \vdash \langle \texttt{Val}\ v.M(es),s\rangle \xrightarrow{\ell} \langle \texttt{Val}\ v.M(es'),s'\rangle} \tag{32}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle e; e_2,s\rangle \xrightarrow{\ell} \langle e'; e_2,s'\rangle} \tag{33}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle \texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2,s\rangle \to \langle \texttt{if}\ (e')\ e_1\ \texttt{else}\ e_2,s'\rangle} \tag{34}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle e \cdot es,s\rangle \ [\xrightarrow{\ell}]\ \langle e' \cdot es,s'\rangle} \qquad \frac{P \vdash \langle es,s\rangle \ [\xrightarrow{\ell}]\ \langle es',s'\rangle}{P \vdash \langle \texttt{Val}\ v \cdot es,s\rangle \ [\xrightarrow{\ell}]\ \langle \texttt{Val}\ v \cdot es',s'\rangle} \tag{35}$$

Fig. 2. Subexpression reduction rules. We define $g(\ell,D) = D$, if $\ell = -$; otherwise $g(\ell,D) = \ell$.

$$\frac{\mathit{new\text{-}Addr}\ h = a \quad P \vdash C\ \mathit{has\text{-}fields}\ FDTs}{P \vdash \langle \mathtt{new}\ C, (h,l)\rangle \xrightarrow{-} \langle \mathit{addr}\ a, (h(a \mapsto\ (C,\ \mathit{init\text{-}fields}\ FDTs)), l)\rangle} \tag{36}$$

$$\frac{\mathit{hp}\ s\ a = (D, fs) \quad P \vdash D \preceq^*\ C}{P \vdash \langle \mathtt{Cast}\ C\ (\mathit{addr}\ a), s\rangle \xrightarrow{-} \langle \mathit{addr}\ a, s\rangle} \tag{37}$$

$$P \vdash \langle \mathtt{Cast}\ C\ \mathit{null}, s\rangle \xrightarrow{-} \langle \mathit{null}, s\rangle \tag{38}$$

$$\frac{\mathit{lcl}\ s\ V = v}{P \vdash \langle \mathtt{Var}\ V, s\rangle \xrightarrow{-} \langle \mathtt{Val}\ v, s\rangle} \tag{39}$$

$$P \vdash \langle V := \mathtt{Val}\ v, (h,l)\rangle \xrightarrow{-} \langle \mathit{unit}, (h, l(V \mapsto\ v))\rangle \tag{40}$$

$$\frac{\mathit{binop}\ (\mathit{bop}, v_1, v_2) = v}{P \vdash \langle \mathtt{Val}\ v_1 \ll \mathit{bop} \gg \mathtt{Val}\ v_2, s\rangle \xrightarrow{-} \langle \mathtt{Val}\ v, s\rangle} \tag{41}$$

$$\frac{\mathit{hp}\ s\ a = (C, fs) \quad fs(F,D) = v}{P \vdash \langle \mathit{addr}\ a.F\{D\}, s\rangle \xrightarrow{-} \langle \mathtt{Val}\ v, s\rangle} \tag{42}$$

$$\frac{\mathit{hp}\ a = (C, fs)}{P \vdash \langle \mathit{addr}\ a.F\{D\} := \mathtt{Val}\ v, (h,l)\rangle \xrightarrow{-} \langle \mathit{unit}, (h(a\ \mapsto\ (C, fs((F,D)\ \mapsto\ v))), l)\rangle} \tag{43}$$

$$\frac{\mathit{hp}\ s\ a = (C, fs) \quad P \vdash\ C\ \mathit{sees}\ M :\ Ts \to T = (\mathit{pns},\ \mathit{body})\ \mathit{in}\ D \quad |vs| = |pns| \quad |Ts| = |pns|}{P \vdash \langle \mathit{addr}\ a.M(\mathit{map}\ \mathtt{Val}\ vs), s\rangle \xrightarrow{-} \langle \mathit{blocks}_D(\mathit{this} \cdot \mathit{pns},\ \mathit{Class}\ D \cdot Ts,\ \mathit{addr}\ a \cdot vs,\ \mathit{body}), s\rangle} \tag{44}$$

$$P \vdash \langle \{V : T;\ \mathtt{Val}\ u\}_D, s\rangle \xrightarrow{D} \langle \mathtt{Val}\ u, s\rangle \tag{45}$$

$$P \vdash \langle \{V : T;\ V := \mathtt{Val}\ v;\ \mathtt{Val}\ u\}_D, s\rangle \xrightarrow{D} \langle \mathtt{Val}\ u, s\rangle \tag{46}$$

$$P \vdash \langle \mathtt{Val}\ v;\ e_2, s\rangle \xrightarrow{-} \langle e_2, s\rangle \tag{47}$$

$$P \vdash \langle \mathtt{if}(\mathit{true})\ e_1\ \mathtt{else}\ e_2, s\rangle \xrightarrow{-} \langle e_1, s\rangle \tag{48}$$

$$P \vdash \langle \mathtt{if}(\mathit{false})\ e_1\ \mathtt{else}\ e_2, s\rangle \xrightarrow{-} \langle e_2, s\rangle \tag{49}$$

$$P \vdash \langle \mathtt{while}(b)\ c, s\rangle \xrightarrow{-} \langle \mathtt{if}(b)\ (c;\ \mathtt{while}(b)\ c)\ \mathtt{else}\ \mathit{unit}, s\rangle \tag{50}$$

Fig. 3. Expression reduction

$$\frac{hp\ s\ a\ =\ (D, fs) \quad \neg\, P \vdash D \preceq^* C}{P \vdash \langle \texttt{Cast}\ C(addr\ a), s\rangle \xrightarrow{-} \langle \texttt{THROW ClassCastException},\ s\rangle} \tag{51}$$

$$P \vdash \langle null.F\{D\}, s\rangle \xrightarrow{-} \langle \texttt{THROW NullPointerException},\ s\rangle \tag{52}$$

$$P \vdash \langle null.F\{D\} := \texttt{Val}\ v, s\rangle \xrightarrow{-} \langle \texttt{THROW NullPointerException},\ s\rangle \tag{53}$$

$$P \vdash \langle null.M(map\ \texttt{Val}\ vs), s\rangle \xrightarrow{-} \langle \texttt{THROW NullPointerException},\ s\rangle \tag{54}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle \texttt{throw}\ e, s\rangle \xrightarrow{\ell} \langle \texttt{throw}\ e', s'\rangle} \tag{55}$$

$$P \vdash \langle \texttt{throw}\ null, s\rangle \xrightarrow{-} \langle \texttt{THROW NullPointerException}, s\rangle \tag{56}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle \texttt{try}\ e\ \texttt{catch}\ (C\ V)\ e_2, s\rangle \xrightarrow{\ell} \langle \texttt{try}\ e'\ \texttt{catch}\ (C\ V)\ e_2, s'\rangle} \tag{57}$$

$$P \vdash \langle \texttt{try}\ \ \texttt{Val}\ v\ \texttt{catch}\ (C\ V)\ e_2, s\rangle \xrightarrow{-} \langle \texttt{Val}\ v, s\rangle \tag{58}$$

$$\frac{hp\ s\ a\ =\ (D, fs) \quad P \vdash\ D\ \preceq^*\ C}{P \vdash \langle \texttt{try THROW}\ a\ \texttt{catch}\ (C\ V)\ e_2, s\rangle \xrightarrow{-} \langle \{V : Class\ C;\ V := addr\ a;\ e_2\}, s\rangle} \tag{59}$$

$$\frac{hp\ s\ a\ =\ (D, fs) \quad \neg\, P \vdash\ D\ \preceq^*\ C}{P \vdash \langle \texttt{try THROW}\ a\ \texttt{catch}\ (C\ V)\ e_2, s\rangle \xrightarrow{-} \langle \texttt{Throw}\ a, s\rangle} \tag{60}$$

Fig. 4. Exceptional expression reduction

$$P \vdash \langle \texttt{Cast}\ C\ (\texttt{throw}\ e), s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{61}$$

$$P \vdash \langle V := \texttt{throw}\ e, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{62}$$

$$P \vdash \langle \texttt{throw}\ e.F\{D\}, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{63}$$

$$P \vdash \langle \texttt{throw}\ e.F\{D\} := e_2, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{64}$$

$$P \vdash \langle \texttt{Val}\ v.F\{D\} := \texttt{throw}\ e, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{65}$$

$$P \vdash \langle \texttt{throw}\ e\ \ll bop \gg\ e_2, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{66}$$

$$P \vdash \langle \texttt{Val}\ v_1\ \ll bop \gg\ \texttt{throw}\ e, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{67}$$

$$P \vdash \langle \{V : T;\ \texttt{Throw}\ a\}_D, s\rangle \xrightarrow{D} \langle \texttt{Throw}\ a, s\rangle \tag{68}$$

$$P \vdash \langle \{V : T; V := \texttt{Val}\ v;\ \texttt{Throw}\ a\}_D, s\rangle \xrightarrow{D} \langle \texttt{Throw}\ a, s\rangle \tag{69}$$

$$P \vdash \langle \texttt{throw}\ e.M(es), s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{70}$$

$$P \vdash \langle \texttt{Val}\ v.M(map\ \texttt{Val}\ vs\ @\ (\texttt{throw}\ e\ \cdot\ es')), s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{71}$$

$$P \vdash \langle \texttt{throw}\ e;\ e_2, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{72}$$

$$P \vdash \langle \texttt{if}(\texttt{throw}\ e)\ e_1\ \texttt{else}\ e_2, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{73}$$

$$P \vdash \langle \texttt{throw}(\texttt{throw}\ e), s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{74}$$

Fig. 5. Exception propagation

$$\frac{P \vdash \langle es, s\rangle \ [\xrightarrow{\ell}] \ \langle es', s'\rangle}{P \vdash \langle D.M(es), s\rangle \xrightarrow{\ell} \langle D.M(es'), s'\rangle} \tag{75}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle e[e_2], s\rangle \xrightarrow{\ell} \langle e'[e_2], s'\rangle} \tag{76}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle (\texttt{Val}\ v)[e], s\rangle \xrightarrow{\ell} \langle (\texttt{Val}\ v)[e'], s'\rangle} \tag{77}$$

$$P \vdash \langle D.M(map\ \texttt{Val}\ vs\ @\ (\texttt{throw}\ e\ \cdot\ es')), s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{78}$$

$$P \vdash \langle (\texttt{throw}\ e)[e'], s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{79}$$

$$P \vdash \langle e'[\texttt{throw}\ e], s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{80}$$

Fig. 6. Subexpression reduction and exception propagation rules for Jinja+.

$$\frac{P \vdash\ D \textit{ has-static } M :\ Ts \to T = (pns,\ body) \qquad |vs| = |pns| \qquad |Ts| = |pns|}{P \vdash \langle D.M(map\ \texttt{Val}\ vs), s\rangle \xrightarrow{-} \langle blocks_D(pns, Ts, vs,\ body), s\rangle} \tag{81}$$

$$\frac{n \geq 0,\ \textit{new-Addr}\ h = a}{P \vdash \langle \texttt{new}\ \tau\texttt{[}intg(n)\texttt{]},\ (h,l)\rangle \to \langle addr\ a, (h(a \mapsto initArr(\tau, n)), l)\rangle} \tag{82}$$

$$P \vdash \langle null.F\{D\}, s\rangle \xrightarrow{-} \langle \texttt{THROW NullPointerException},\ s\rangle \tag{83}$$

$$\frac{n < 0}{P \vdash \langle \texttt{new}\ \tau\texttt{[}intg(n)\texttt{]},\ (h,l)\rangle \xrightarrow{-} \langle \texttt{THROW NegativeArraySizeException},\ (h,l)\rangle} \tag{84}$$

$$\frac{h\ a = (\tau, m, t),\ 0 \leq n < m,\ h\ t(n) = v}{P \vdash \langle (addr\ a)\texttt{[}intg\ n\texttt{]},\ (h,l)\rangle \xrightarrow{-} \langle Val\ v,\ (h,l)\rangle} \tag{85}$$

$$\frac{h\ a = (\tau, m, t),\ \neg(0 \leq n < m),}{P \vdash \langle (addr\ a)\texttt{[}intg\ n\texttt{]},\ (h,l)\rangle \xrightarrow{-} \langle \texttt{THROW IndexOutOfBoundsException},\ (h,l)\rangle} \tag{86}$$

$$\frac{h\ a = (\tau, m, t),}{P \vdash \langle (addr\ a)\texttt{.lenght},\ (h,l)\rangle \xrightarrow{-} \langle intg\ m,\ (h,l)\rangle} \tag{87}$$

$$\frac{h\ a = (\tau, m, t),\ 0 \leq n < m,\ isOfType(v, \tau),\ t' = arrayUpdate(t, n, v)}{P \vdash \langle (addr\ a)\texttt{[}intg\ n\texttt{]}\ \texttt{:=}\ Val\ v,\ (h,l)\rangle \xrightarrow{-} \langle unit,\ (h(a \mapsto (\tau, m, t')), l)\rangle} \tag{88}$$

$$\frac{h\ a = (\tau, m, t),\ \neg(0 \leq n < m),}{P \vdash \langle (addr\ a)\texttt{[}intg\ n\texttt{]}\ \texttt{:=}\ Val\ v,\ (h,l)\rangle \xrightarrow{-} \langle \texttt{THROW IndexOutOfBoundsException},\ (h,l)\rangle} \tag{89}$$

$$\frac{h\ a = (\tau, m, t),\ 0 \leq n < m,\ \neg isOfType(v, \tau),}{P \vdash \langle (addr\ a)\texttt{[}intg\ n\texttt{]}\ \texttt{:=}\ Val\ v,\ (h,l)\rangle \xrightarrow{-} \langle \texttt{THROW ArrayStoreException},\ (h,l)\rangle} \tag{90}$$

Fig. 7. (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 $body$.