A Proof Technique for Noninterference In Open Systems An extended version

Enrico Scapin

University of Trier, Germany
scapin@uni-trier.de

Abstract. In [11], a framework has been proposed which allows tools that can check noninterference properties but a priori cannot deal with cryptography (probabilities and polynomially bounded adversaries) to establish cryptographic indistinguishability properties, such as privacy properties, for Java programs.
As for checking noninterference, many program analysis tools can only deal with closed Java programs. The systems to be analyzed are, however, often open: they interact with a network or use some libraries which are not necessarily trusted and, hence, are not part of the code to be analyzed. Therefore, as part of the framework, a *proof technique* was proposed to reduce the problem of checking noninterference in an open system to checking noninterference for a single (almost) closed system.
In this work, we extend the syntax and semantics of Jinja+, the language the framework is stated for, with java-interfaces, abstract classes, and strings. Consequently, we show that, except for the proof technique for noninterference in open systems, all definitions and results of the framework carry out easily to the extended language. Nonetheless, regarding the proof technique, non-trivial modifications have been required in order to model the exchange of data between the system and the environment, when also strings are involved.

1 Introduction

In [11], Küsters, Truderung, and Graf considered the problem of establishing security guarantees – in particular, computational indistinguishability – for Java-like programs that use cryptographic primitives, for instance public key encryption. They proposed a framework, called "The CVJ Framework" (Cryptographic Verification of Java programs), which allows tools that can check standard noninterference properties [4, 17] but a priori cannot deal with cryptography, specifically probabilities and polynomially bounded adversary, to establish cryptographic indistinguishability properties, such as privacy properties. In particular, two systems S_1 and S_2 are defined to be computationally indistinguishable if no probabilistic polynomially bounded environment is able to distinguish, with more than negligible probability, whether it interacts with S_1 or S_2.

The framework combines techniques from program analysis and cryptography, more specifically, universal composability [3, 16, 8, 13], a well-established approach for the modular security analysis of cryptographic protocols. The crucial problem in this scenario is that existing program analysis tools for noninterference can not deal with cryptography since they do not consider probabilities of events or do not model the common adversary model in cryptography, i.e., the polynomially bounded adversary. They

indeed prove security properties with respect to unbounded adversary: for instance if a message is encrypted and the ciphertext is given to the adversary, the tools consider this to be an illegal information flow (unless the ciphertext is *declassified* [19, 18, 14]), because a computational unbounded adversary could decrypt the message.

The approach is then to first check noninterference properties for the Java program to be analyzed where cryptographic operation (such as encryption) are performed within so-called *ideal functionalities*. Ideal functionalities trace out the operational behavior of communication protocols where party can exchange data in a perfect, ideal setting in such a way that they all achieve the desired protocol outcome. Techniques from *simulation-based security* [2, 16, 8, 9] are used to prove security of real cryptographic protocols by comparing it with their corresponding ideal functionalities: a real protocol R realizes and ideal protocol F if there exists a simulator S such that R and $S \cdot F$ are computational indistinguishable. What it is relevant with respect to the framework is that such functionalities typically provide guarantees even in face of unbounded adversaries and can often be formulated without probabilistic operation. Hence, tools that a priori can not deal with cryptography can carry out analyses of noninterference properties without risk of false positives due to the use of cryptographic primitives.

At the core of the CVJ framework, there are results linking the notion of (termination-insensitive) noninterference [17] with the notion of cryptographic indistinguishability: in order to assert that two systems using cryptographic operations are computational indistinguishable, it is enough to show that these systems are noninterferent when the cryptographic operations are replaced by their corresponding ideal functionalities. That is, noninterference of a system using ideal cryptographic functionalities implies computational indistinguishability of the system using their realizations.

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.

In [11] only an ideal functionality for public-key encryption has been proposed and it has been shown that it can be realized by any IND-CCA2-secure public-key encryption scheme, a standard security notion for such schemes (see, e.g., [1]). This functionality does not support reasoning about corruption and also it does not support a public-key infrastructure (PKI). Therefore, in [10], the CVJ framework has been further instantiated with more suitable ideal functionalities which commonly occur in cryptographic applications and to provide realizations of these functionalities based on standard cryptographic assumptions: while so far similar functionalities have been considered in the cryptographic literature only based on Turing machine models (see, e.g., [16, 3, 12]), now they has been directly typed out in *Java* in such a way that these functionalities can actually be used to analyze Java programs. In particular, in [10] have been proposed ideal functionalities for public-key encryption, digital signature (both supporting *static* corruption and a public-key infrastructure), private symmetric encryption, and nonce generation. Furthermore, it has been shown that these functionalities can be realized

using standard cryptographic schemes and assumptions: more concretely, IND-CCA2-secure public-key encryption schemes for public-key encryption and private symmetric encryption, whereas UF-CMA-secure for digital signature and freshness for nonce generation.

I-noninterference. As to checking non-interference, many program analysis tools can only deal with closed Java programs. The systems to be analyzed are, however, often open: they interact with a network or use some libraries which are not necessarily trusted and, hence, are not part of the code to be analyzed; instead, they are considered as part of the environment, with unspecified behavior. Therefore, [11] intruduces the notion of noninterference in an open system (a.k.a *I-noninterference*), i.e., in a system not completely defined: An open system S is noninterferent if for each environment E this system can be composed with, the resulting close system $S \cdot E$ is also noninterferent. As part of the framework, a *proof technique* was also proposed to reduce the problem of checking noninterference in an open system to checking noninterference for a single (almost) closed system. Technically, this result shows how to construct, for an open system S, a family of environments $\tilde{E}_{\vec{u}}$ parametrized by an input sequence $\vec{u}$, such that S is noninterferent if and only if S composed with $\tilde{E}_{\vec{u}}$ is noninterferent for all $\vec{u}$. Importantly, the latter property can be verified using existing tools for program analysis. These techniques have been used in all the case studies related to the framework so far, including in the verification of a deployed *cloud storage system* [10] making use of all the cryptographic primitives listed above, but they are rather general, and hence, relevant beyond these case studies.

Our contribution. The framework is formulated for a language called Jinja+ and is proven w.r.t. the formal semantics of this language. Jinja+ is a Java-like language that extends the language Jinja [7] with, among others, arrays, the type `byte`, and the `abort` primitive. In this work, we further extend its syntax and semantics with: (a) java-interfaces, (b) abstract classes, (c) strings. Except for one result (discussed below), all definitions and results of the framework carry out easily to the extended language. That is, the new types of values and the new rules of the augmented small-step semantics do not affect the proofs in a significant way.

One result, however, namely the proof of reducing the problem of noninterference in an open system to checking noninterference for a single (almost) closed system, required non-trivial modifications to model the exchange of data between the system and the environment when also strings are involved. In particular, the exchange of data through string references introduces subtle changes in the original result and, technically, invalidates the main assumption the result is based on, i.e., the separation between the state of the system and the state of the environment. Therefore, we extend the construction of $\tilde{E}_{\vec{u}}$ to handle exchange of string references. Furthermore, relying on the fact that the Java (Jinja+) strings are *immutable*, we relax the state separation assumption and adopt the proof in a non-trivial way to work with the new (relaxed) assumption.

Based on this premise, we then reshaped the proof technique for proving noninterference in open systems taking into account string references, too.

Structure of the paper. The language Jinja+ is introduced and extended in Section 2. In Section 3, the CVJ framework is briefly explained, highlighting the section related

to noninterference in open systems. In Section 4, we extend the proof technique for proving I-noninterference: after recalling the result already stated in [11] for primitive types, we extend this result to the communication through strings and we rearrange the communication through simple objects, arrays, and exceptions more accurately and in accord with the aforementioned extension. The proof of the main result can be found in Appendix A.2, as long as the full small-step semantics of Jinja+ (Appendix B).

2 Jinja+: A Java-like language

The CVJ framework is stated for a Java-like language which we call *Jinja+*. Jinja+ is based on *Jinja* [7] and extends this language with some additional features that are useful or needed in the context of our framework.

Jinja+ covers a rich subset of Java, including (java-)interfaces, abstract and concrete classes, inheritance, (static and non-static) fields and methods, the primitive types `int`, `boolean`, and `byte` (with the usual operators for these types), strings, arrays, exceptions, and field/method access modifiers, such as `public`, `private`, and `protected`. Among the features of Java that are *not* covered by Jinja+ are: concurrency and reflection.

We now present the syntax and the semantics of Jinja+.

2.1 Syntax

As already mentioned, the subset of Java we consider is based on the language Jinja [7]. Expressions in Jinja are constructed recursively and include: (a) creation of a new object, (b) casting, (c) literal values (constants) of types boolean and int, (d) **`null`**, (e) binary operations, (f) variable access and variable assignment, (g) field access and field assignment, (h) method call, (i) blocks with locally declared variables, (j) sequential composition, (k) conditional expressions, (l) while loop, (m) exception throwing and catching. As a basis of the formal results of the CVJ framework, in [11] Jinja has been extended 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.

For completeness, we now recall these extensions as they have been presented in [11]. Then, in the next subsection, we further develop the language.

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

Jinja comes equipped with a type system and a notion of well-typed programs. We follow the convention of [11], where only well-typed programs are considered.

Primitive types. The Jinja language, as specified in [7], 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 [15].

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 [15], 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. Extending Jinja with static fields 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.

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.

Access modifiers, final classes, and `throws` clauses. As for java-interfaces and abstract classes, for these three extensions 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).

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

2.2 A further extension: java-interfaces, abstract classes, and strings.

Starting from the aforementioned extension, we further develop the language including: a) java-interfaces, b) abstract classes, c) *strings*. We notice that the concept of *java-interface* follows the abstract type that is indeed used to specify interfaces in Java: it is not to be confused with the concept of interface presented in the CVJ framework (see Section 3).

We now present the *abstract syntax* of a Jinja+ system and then we describe the above-mentioned extensions. A *system* or a *program* is a triple of three lists: a list of

java-interface declarations, a list of *abstract class* declarations and a list of *concrete class* declarations. Each declaration consists of the name of the java-interface/class and the java-interface/class itself. An *java-interface* consists of a list of its direct superinterfaces (optionally), a list of its constant declarations, and a list of method signatures. A *constant declaration* consists of a type, a constant name and a literal of the appropriate type (the value of the constant). A *method signature* consists of the method name, the formal parameter names and types, and the result type. An *abstract class* consists of the name of its direct superclass (optionally), a list of the implemented java-interfaces (optionally), a list of field declarations, a list of method signatures (commonly called abstract methods) and a list of method declarations. A *field declaration* consists of a type and a field name. A *method declaration* consists in a method signature and in an expression (the method body). Note that there is no `return` statement, as a method body is an expression; the value of such an expression is returned by the method. A *concrete class* is defined as an abstract class without the list of method signatures, i.e., all methods contain the body. In the above definitions we assume that different syntactic categories (java-interfaces, classes, constants, fields and methods) have different names.

Java-Interfaces and Abstract Classes. We assume that java-interfaces and abstract classes are provided by a compiler that, first, ensures that the policies expressed by these clauses are respected (e.g., a concrete class implementing an inferface indeed implements all its methods) and then produces Jinja+ code without them. Regarding the Jinja+ rule (see Appendix B), we extend the convention used in [7] that symbols C and D denote (concrete or abstract) classes to denoting java-interfaces, too. According to this convention, there are a few changes in the interpretation of the expressions and predicates of the smallstep semantics rules:

- In the expression *new* C corresponding to the creation of a new object we assume that a compiler already enforced C to be a concrete class. On the other hand side, we require that the predicate $P \vdash C$ *has-fields* $FDTs$ used in the object creation rule (Rule 16) collects information about the fields both in the class and in the java-interface hierarchy.
- In the expression *Cast* C now C can be either a (concrete or abstract) class or an java-interface. Therefore, we extend the meaning of the predicate $P \vdash D \preceq^* C$ (used in rules 17, 18, 31, 39 and 40): It means D is a subclass of C if C is a class, whereas D implements C if C is an java-interface.
- In the expression $e.F\{D\}$ (field access) D can now be either the class or the java-interface where F is declared (in the latter case F is defined as a constant). On the other hand side, in the expression $e.F\{D\} := e_2$ (field assignment) D can only be the (abstract or concrete) class where F is declared. However, the rules where these expressions are evaluated (Rule 3 and 4) remain unchanged.
- In the expression `try` e_1 `catch` $(C\,V)\ e_2$ we assume that a compiler already enforced C to be a concrete class (which must extend the class $Throwable$). Therefore in rules 39 and 40, the predicate $P \vdash D \preceq^* C$ is indeed always interpreted as D is a subclass of C.
- The predicate $P \vdash C$ *sees* $M: Ts \to T = (pns, body)$ *in* D, used in the rule 24 is supposed to look up for the method declaration of M in the class hierarchy and

therefore now also in abstract classes but obviously not in the java-interfaces (since they contain only method signatures).

The interpretation of the other expressions and rules remain the same as in [7].

Strings. We extend the language *Jinja* [7] with strings. We introduce a new value *litS* detoning a *string literal*, i.e., a quoted sequence of characters representing a string value within the source code. In the heap, we represent a string as a pair consisting on an array of the characters in the string literal and its length. The extension of the (small-step) semantic rules to deal with strings is quite straightforward and can be found in Figure 13 of Appendix B.

2.3 Semantics

Following [7], we briefly sketch the small-step semantics of Jinja. The full set of rules, including those for Jinja+ (see the next subsection) can be found in Appendix B.

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

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

We assume that Jinja+ programs have unbounded memory. The reason for this modeling choice is that Jinja+ is used to formulate a framework for the cryptographic verification of Java-like programs, where the formal foundation for the security notions adopted are based on *asymptotic* security. This kind of security definitions only makes sense if the memory is not bounded, since the security parameter grows indefinitely; see Section 3 and, foremost, [11] for the details.

Randomized programs. So far, we have considered deterministic programs. We will also need to consider randomized programs in our framework. For this purpose, Jinja+ programs may use the primitive `randomBit()` that returns a random bit each time it is used. Jinja+ programs that do not make use of `randomBit()` are (called) *deterministic*, and otherwise, they are called *randomized*.

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

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

to their initial values (if the initial value for a static variable is not specified in the program, the default initial value for its type is used).

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

We note that all references appearing in each configuration $\langle e,(l,h)\rangle$ are in the domain of the heap h. This is because at the beginning of the run we have $h_0 = \emptyset$ and every time a new object is created, this object and its reference are added to the heap of the state, as described by rules 16, 62, 71, 72 and 76 of Jinja+.

The small-step semantics of Jinja+ provides a natural measure for the *length of a run* of a program, and hence, the runtime of a program. The *length of a run of a deterministic program* is the number of steps taken using the rules of the small-step semantics. For a run r of a program P containing some subprogram S (a subset of classes of P), we define the *number of steps performed by S* or the *number of steps performed in the code of S* in the expected way. To define this notion, we keep track of the origin of (sub)expressions, i.e., the class they come from. If a rule is applied on a (sub)expression that originates from the class C, we label this step with C and count this as a step performed in C (see Appendix B for details).

3 The CVJ Framework

We briefly recall the CVJ framework (Cryptographic Verification of Java programs) which has been introduced in [11] and extended in [10].

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.

While in [11] the CVJ framework only supports a basic version of public-key encryption, in [10] this framework has been further instantiated to support public-key encryption and digital signatures, both with corruption and a public-key infrastructure, as well as (private) symmetric encryption and nonce generation. Since these cryptographic primitives are very common in security-critical applications, these extensions make the framework much more widely applicable.

The extension of Jinja+ with java-interfaces and abstract classes does not affect any of the definitions and theorems presented in [11] since, as already explained in subsection 2.1, we assume a compiler ensures their policies and then produces a Jinja+ code without them. On the other hand side, the extension of Jinja+ with strings demands an adaptation of the prooftechnique for proving I-noninterference: the code of $\tilde{E}_{\vec{u}}^{I_E}$ has to be enriched with two more static methods in order to handle the communication through strings, too.

The definitions and theorems stated below are somewhat simplified and informal, but should suffice to grasp the essence of the framework. We refer the reader to [11] for full details.

Indistinguishability. An *interface* I (not to be confused with the concept of java-interfaces presented in the Jinja+ language) 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 $|\mathsf{Prob}\{P_1(\eta) \rightsquigarrow \mathtt{true}\} - \mathsf{Prob}\{P_2(\eta) \rightsquigarrow \mathtt{true}\}|$ is a negligible function in the security parameter η.[1]

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

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

More formally, let F and R be probabilistic polynomially bounded systems which implement the same interface I_{out} and use the same interface I_E, except that in addition F may use some interface I_S provided by a simulator. Then, we say that R *realizes F w.r.t. I_{out}*, written $R \leq^{I_{out}} F$ or simply $R \leq F$, if there exists a probabilistic polynomially bounded system S (the simulator) such that $R \approx^{I_{out}}_{\mathsf{comp}} S \cdot F$. As shown in [11], $\leq$ 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) [11]). *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.*

The proof of this Theorem can be found in [11].

Noninterference. The (standard) noninterference notion for confidentiality [4] 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,

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

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.

Definition 2 (Noninterference in an open system [11]). *Let I be an interface and let $S[\vec{x}]$ be a (not necessarily closed) deterministic system with a security parameter, $S : I$, and high and low variables. 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, for convenience, the variable* `result` *declared in E is considered to be a 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 [11] 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, [11]). *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 [6, 5], 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 $S[\vec{x}]$. An extended and revisited version of this theorem is presented in Section 4.

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, [11]). *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

$(\forall\ \vec{a}_1, \vec{a}_2 \colon P[\vec{a}_1] \cdot R \approx^{J}_{\mathsf{comp}} P[\vec{a}_2] \cdot R)$ it suffices to prove I-noninterference for $P[\vec{x}] \cdot F$, i.e., the system where R is replaced by the ideal counterpart F (the ideal cryptographic library). The ideal functionality F, which in our case will model cryptographic primitives in an ideal way, can typically be formulated without probabilistic operations and also the ideal primitives specified by F will be secure even in presence of unbounded adversaries. Therefore, the system $P[\vec{x}] \cdot F$ can be analyzed by standard tools that a priori cannot deal with cryptography (probabilities and polynomially bounded adversaries).

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

4 Extending the Proof Technique for proving I-Noninterference

In [11], a proof technique has been introduced for proving (termination-insensitive) non-interference for open systems, i.e., systems not completely defined (e.g., programs whose external libraries have been stripped off). Technically, it has been show that in order to check non-interference for open systems according to Definition 2 it is sufficient to consider only a very restricted class of environments, rather than all environments. The input these environments give to the system they interact with is fixed for every environment and does not depend on the output the environment got from the system. In fact, the environments in this class are all almost identical, they only differ in the input sequence they use. Now, the analysis a tool performs often ignores or can ignore specific values encoded in the program, such as the input sequence. So, if such an analysis establishes non-interference for a system and a fixed environment in the considered class, then this implies that non-interference holds for all environments in this class. By this proof technique, it then follows that non-interference holds for all environments, as required by Definition 2.

A simple and a non trivial case studies have been presented in [11] and [10] respectively, with the aim to demostrate that using our proof techniques, the tool Joana, which is designed to check non-interference for closed systems, can now deal with (a relevant class of) open systems as well.

We now first recall the proof technique when only pure data (primitive types) is exchanged between the system and the environment as it has been presented in [11]. Then, we extend the proof technique to the case where the communication takes place through strings, simple objects, arrays, and exceptions.

4.1 Communication through Primitive Types Only

In this section, we assume that a system S communicates with an environment E only through static functions with primitive types. More precisely, we consider programs S such that (1) method `main` is defined in S and (2) $I_E \vdash S$, for some interface I_E, where all methods are static, use primitive types only (for simplicity of presentation we will consider only the type `int`), and have empty `throws` clause. We will consider indistinguishability w.r.t. the empty interface (i.e. environments we consider do not directly call S). The above assumptions will allow us to show, in the proof of Theorem 4, that E and S do not share any references: their states are in this sense disjoint.

We now define the class of environments mentioned at the beginning of this section. We then show that to establish I-non-interference, it is enough to consider only those environments (see Theorem 4).

For a finite sequence $\vec{u} = u_1, \ldots, u_n$ of values of type `int`, we denote by $\tilde{E}^{I_E}_{\vec{u}}$ the following system. The environment $\tilde{E}^{I_E}_{\vec{u}}$ contains two static methods: `untrustedOutput` and `untrustedInput`, as specified in Figure 1. For the sake of the discussion, let S be the system the environment $\tilde{E}^{I_E}_{\vec{u}}$ interacts with. (Note that the definition of $\tilde{E}^{I_E}_{\vec{u}}$ is independent of a specific S. It only depends on I_E.) As we will see below, the method `untrustedOutput` gets all data passed by S to $\tilde{E}^{I_E}_{\vec{u}}$. The method `untrustedInput` determines what the environment passes on to S.

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

The method `untrustedOutput`, depending on the values given by `untrustedInput()`, either ignores its argument or compares its value to the next integer it obtains, again, from `untrustedInput()` and stores the result of this comparison in the (low) variable `result`. The intuition is the following: `untrustedOutput` gets, as we will see below, all the data the environment gets from S. If the two instances of S, $S[\vec{a}_1]$ and $S[\vec{a}_2]$, which the environment tries to distinguish, behave differently, then there must be some point where the environment gets different data from the two systems in the corresponding runs, i.e., `untrustedOutput` will be invoked with different values for x, say the values x takes at this point are b_1 and b_2, respectively. By choosing an appropriate $\vec{u}$, this can be detected by `untrustedOutput`: $\vec{u}$ should be defined in such a way that the method `untrustedInput()` returns 1 at this point and that the value `untrustedInput()` returns next equals b_1, say (b_2 would also work). Then, in the run of the environment with $S[\vec{a}_1]$ the variable `result` will be assigned 1 and in the run with $S[\vec{a}_2]$ it will be assigned 0. Hence, the environment successfully distinguished $S[\vec{a}_1]$ and $S[\vec{a}_2]$.

Finally, for every method declaration m in I_E, the system $\tilde{E}^{I_E}_{\vec{u}}$ contains the implementation of m as illustrated by the example in Figure 2. As we can see, the defined method forwards all its input data to `untrustedOutput` and lets `untrustedInput` determine the returned value.

This completes the definition of $\tilde{E}^{I_E}_{\vec{u}}$. The next theorem (see Appendix A.1 for the proof) states that, to prove I-non-interference, it is enough to consider only environments $\tilde{E}^{I_E}_{\vec{u}}$ for all $\vec{u}$. In this way we need to study only (almost) closed systems, namely

```
class Node {
    int value;
    Node next;
    Node(int v, Node n) { value = v;  next = n; }
}
private static Node list = null;
private static boolean listInitialized = false;
private static Node initialValue()
    { return new Node(u1, new Node(u2, ...)); }
static public int untrustedInput() {
    if (!listInitialized)
        { list = initialValue(); listInitialized = true; }
    if (list==null) return 0;
    int tmp = list.value;
    list = list.next;
    return tmp;
}
static public void untrustedOutput(int x) {
    if (untrustedInput()!=0) {
        result = (x==untrustedInput());
        abort();
    }
}
```

Fig. 1. Implementation of `untrustedInput`, `untrustedOutput` in $\tilde{E}_{\vec{u}}$. We assume that class `Node` not used anywhere else.

```
static public int foo(int x) {
    untrustedOutput(FOO_ID);
    untrustedOutput(x);
    return untrustedInput();
}
```

Fig. 2. $\tilde{E}_{\vec{u}}^{I_E}$: the implementation of a method of I_E with the signature `static public int foo(int x)`, where `FOO_ID` is an integer constant serving as the identifier of this method (we assign a different identifier to every method).

systems that differ in only one expression (line 9). As discussed at the beginning of this section, this restriction is often sufficient for tools are designed to deal with closed systems only.

Theorem 4. *Let I_E be an interfaces with only static methods of primitive argument and return types as introduced above. Let S be a system with high and low variables such that* `main` *is defined in S and $I_E \vdash S$. Then, I-non-interference, for $I = \emptyset$, holds for S if and only if for all sequences $\vec{u}$ as above non-interference holds for $\tilde{E}_{\vec{u}}^{I_E} \cdot S$.*

The proof of this theorem, as long as all the necessary lemmas to demonstrate it, is recalled from [11] in Appendix A.1.

4.2 Communication through Strings, Simple Objects, Arrays, and Exceptions

The result stated in Theorem 4 has been extended in [11] to cover some cases, where E and S exchange information not only through values of primitive types, but also through simple objects, arrays, and throwing exceptions. We now further extend it with strings. We recall that we consider programs S such that (1) method `main` is defined in S, (2) $I_E \vdash S$, for some interface I_E, where all methods are static and uses only the data types listed above, and (3) $I = \emptyset$, that is, we consider indistinguishability w.r.t. the empty interface (i.e. environments do not directly call S).

Strings. We model the exchange of references of type `String` by extending the environment $\tilde{E}_{\vec{u}}^{I_E}$ with two other static methods presented in Figure 3: The method `untrustedOutputString` gets all string references passed by S to $\tilde{E}_{\vec{u}}^{I_E}$, whereas the method `untrustedInputString` determines which string the environment passes on to S. We notice that these methods rely on `untrustedOutput` and `untrustedInput` defined for primitive types and, hence, the sequence $\vec{u} = u_1, \ldots, u_n$ remain the same as defined in Section 4.1.

More specifically, the method `untrustedInputString`, depending on `untrustedInput`, either returns a new string (built also calling `untrustedInput` for each character) or it returns one of the strings previously exchanged between the systems. In particular, each new string returned to S is previously added to a list *stringList* containing all the strings exchanged between the two systems so far.

The method `untrustedOutputString`, besides adding its string argument to *stringList*, forwards to `untrustedOutput` its length, then each one of its characters, and, finally, the result of the comparison for reference equality between its string argument and each element in *stringList*.

The intuition is the following: if two instances of S, $S[\vec{a}_1]$ and $S[\vec{a}_2]$, which the environment tries to distinguish behave differently, then there must be a point in the two runs where the environment gets either two strings with different values or two strings whose references were already been exchanged before (more precisely, at least one of them), but in different points of the two runs. In the former case, it must be that either the length or at least one character of the two strings is different. In the latter case, there must exist two elements at the same position in *stringList*, whose comparison with the two strings the environment received give different results. In any case, `untrustedOutput` will be invoked with different values, say the value x takes at

```
class NodeList {
  public class Node {
    public String entry;
    public Node next;
    public Node(String entry) {
      this.entry = entry;
      this.next=null;
    }
    public Node head, last;
    public void add(String entry) {
      Node newEntry=new Node(entry);
      if (head==null) head=last=newEntry;
      else {last.next=newEntry; last=newEntry;}
    }
}
private static NodeList stringList = null;
static public String untrustedInputString() {
  int choice = untrustedInput();
  if(choice==1){
    int l=untrustedInput(); String s="";
    for(int i=0; i<l; i++)
      s += (char) untrustedInput();
    if(stringList==null) stringList = new NodeList();
    stringList.add(s);
    return s;
  } else if(choice==2){
    if(stringList==null) return "";
    for(NodeList.Node node=stringList.head; node!=null;
     node=node.next)
      if(untrustedInput()==1) return node.entry;
  }
  return "";
}
static void untrustedOutputString(String s) {
  if(stringList==null)
      stringList = new NodeList();
  // values comparison
  untrustedOutput(s.length());
  for (int i = 0; i < s.length(); i++)
      untrustedOutput(s.charAt(i));
  // references comparison
  for(NodeList.Node node=stringList.head; node!=null;
    node=node.next)
     untrustedOutput(s==node.entry ? 1:0);
  stringList.add(s);
}
```

Fig. 3. Implementation of `untrustedInputString` and `untrustedOutputString` in $\tilde{E}_{\vec{u}}$. We notice that these methods rely on methods `untrustedInput` and `untrustedOutput` presented in Figure 1. We assume that class `NodeList` is not used anywhere else.

this point are b_1 and b_2, respectively. As for primitive types, by choosing an appropriate $\vec{u}$, this can be detected by `untrustedOutput`: $\vec{u}$ should be defined in such a way that the method `untrustedInput` returns 1 at this point and that the value `untrustedInput()` returns next equals b_1, say (b_2 would also work). Then, in the run of the environment with $S[\vec{a}_1]$ the variable `result` will be assigned 1 and in the run with $S[\vec{a}_2]$ it will be assigned 0. Hence, the environment successfully distinguished $S[\vec{a}_1]$ and $S[\vec{a}_2]$.

Methods of I_E dealing with strings are implemented as the corresponding methods dealing with primitive values: the arguments are forwarded to $untrustedOutputString$, whereas $untrustedInputString$ determines the returned value.

Simple Objects, Arrays, and Exceptions. In case of communication through simple objects (as introduced below), arrays (either of primitive types, or of strings), and exceptions, some restrictions have to be imposed on I_E and on the program S.

```
29 public static byte[] untrustedInputMessage() {
30  int len = untrustedInput();
31  if (len<0) return null;
32   byte[] returnval = new byte[len];
33   for (int i = 0; i < len; i++)
34    returnval[i] = (byte) untrustedInput();
35   return returnval;
36 }
37 public static void untrustedOutputMessage(byte[] t) {
38  untrustedOutput(t.length);
39  for (int i = 0; i < t.length; i++) {
40   untrustedOutput(t[i]);
41  }
42 }
```

Fig. 4. Implementation of `untrustedInputMessage` and `untrustedOutputMessage` in $\tilde{E}_{\vec{u}}$. We notice that these methods rely on methods `untrustedInput` and `untrustedOutput` presented in Figure 1.

Nevertheless, before introducing these restrinctions, we want to highlight a particular data type essential in the CVJ framework: *byte array*. This data tpye is used in all the cryptographic operations and the transmission of all data among parties, e.g., a client and a server. Therefore, we extended the environment with two other methods presented in Figure 4: The method `untrustedOutputMessage` gets all byte arrays passed by S to $\tilde{E}_{\vec{u}}^{I_E}$, whereas `untrustedInputMessage` determines which byte array the environment passes on to S. As for strings, these methods rely on `untrustedInput` and `untrustedOutput`, respectively, to construct the byte array which has to be returned and to try to distinguish whether two instances of S behave differently.

However, we notice that some restrictions have to be imposed on the system S exchanging byte arrays with the environment. These restrictions guarantee that, although references are exchanged between E and S, the communication resembles exchange of

```
class T extends Exception {};
class T1 extends T {};
class T2 extends T {};
class D { int y; String w; byte[] v; }
class Foo {
 static public String[] foo(int x, String s, byte[] b, D obj) throws T {
  // consume the input:
  untrustedOutput(0x100); // foo id
  untrustedOutput(x);
  untrustedOutputString(s);
  untrustedOutputMessage(b);
  untrustedOutput(obj.hashCode());
  untrustedOutput(obj.y);
  untrustedOutputString(obj.w);
  untrustedOutputMessage(obj.v);
  // decide whether to throw some exceptions:
  if (untrustedInput()==0) throw new T1();
  if (untrustedInput()==0) throw new T2();
  // determine the object to return:
  int length=untrustedInput();
  String[] retStr = new String[length];
  for(int i=0; i<length; ++i)
   retStr[i]=untrustedInputString();
  return retStr;
 }
}
```

Fig. 5. $\tilde{E}_{\vec{u}}$: the implementation of the class `Foo` in I_E with only a method whose signature is **`static public`** `String[] foo(`**`int`** `x, String s,` **`byte`**`[] t, D obj)` **`throws`** `T`, and where `T`, `T1`, `T2`, and `D` are classes in I_E, too. We assume that in Jinja+ (as in Java) each object has an unique identifier provided by the method `hashCode`.

pure data. More precisely, our result works for the following class of systems S. Let the method `main` be defined in S and let I_E be the minimal interface such that $I_E \vdash S$. The assumptions which have to be imposed on I_E are:

E.1. Fields of classes in I_E are non-static and either of primitive types, or strings, or *simple objects*, or arrays of primitive types. Simple objects denote objects of classes defined in I_E.

E.2. Methods of classes in I_E are static. Their arguments and return values may be either of primitive types, or strings, or simple objects, or arrays of primitive types.

E.3. Exceptions thrown by methods of classes in I_E are either standard system exceptions or exceptions defined in I_E.

Let us notice that this only restricts the way S uses the environment; an environment that implements the interface I_E can have arbitrary fields, methods, and exceptions. For such environments we can construct a fixed $\tilde{E}^{I_E}_{\vec{u}}$ such that is enough to consider only these systems (for all $\vec{u}$). This system consists of the static methods `untrustedInput` and `untrustedOutput` for primitive types, `untrustedInputString` and `untrustedOutputString` for values of type `String`, `untrustedInputMessage` and `untrustedOutputMessage` for values of type `byte[]`, and, for every class C of I_E, of the declaration of C with the implementation of its (static) methods as in the example given in Figure 5. This example illustrates the case when either an exception or an array of strings is, respectively, thrown or returned. In particular, in the former case a fresh exception is created and then thrown (e.g., line 17), whereas in the latter case a fresh array is created (line 21), its elements are filled by `untrustedInputString` and, finally, it is returned.

On the other hand side, the assumptions which have to be imposed on S are the following:

S.1. Whenever a simple object or an array (i.e. the reference to a simple object or to an array, respectively) is passed to the environment, this reference is not used by S afterwards. This property can be easily guaranteed by a syntactical restriction to pass only fresh copies of these reference to the environment.

S.2. Whenever a method of I_E returns a reference r different from string, the system S is only allowed to immediately produce a fresh copy of r and not to use r afterwards.

S.3. For every try-catch statement in S of the form

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

if C or a subclass of C is listed in the `throws` clause of some method in I_E (and thus this statement may potentially catch an exception thrown by E), then again S is only allowed to immediately produce a fresh copy of r and not to use r afterwards.

Fresh copies of simple objects and arrays may be provided by support methods which accept as argument the reference of the object to be cloned and return a new object of the same type, where either each field, in case of simple objects, or each element, in case of arrays, has been cloned. An example of these methods is given in Figure 6, where copies of byte arrays, arrays of strings, and simple objects (whose classes have been defined in Figure 5), have to be produced according to the restrictions discussed

```
static byte[] copyOf(byte[] b){
 if (b==null) return null;
 byte[] copy = new byte[b.length];
 for (int i = 0; i < b.length; i++)
  copy[i] = b[i];
 return copy;
}
static String[] copyOf(String[] s){
 if (s==null) return null;
 String[] copy = new String[s.length];
 for (int i = 0; i < s.length; i++)
  copy[i] = s[i];
 return copy;
}
static D copyOf(D obj){
 if (obj==null) return null;
 D copyObj = new D();
 copyObj.y = obj.y;
 copyObj.w = obj.w;
 copyObj.v = copyOf(obj.v);
 return copyObj;
}
static T copyOf(T e){
 if (e==null) return null;
 if (e instanceof T1) return new T1();
 if (e instanceof T2) return new T2();
 return new T();
}
void bar(){
 int x=89; byte[] b={0x50,0x6f,0x6d,0x65};
 String s="The␣Magic␣Words␣are␣";
 D obj = new D(); obj.y=144;
 obj.w="Squeamish␣Ossifrage";
 obj.v=new byte[]{0x72,0x61,0x6e,0x63,0x65};
 String[] retObj;
 try{
   retObj = copyOf(Foo.foo(x, s, copyOf(b), copyOf(obj)));
 } catch (T excp) {
   T myExcp=copyOf(excp);
   myExcp.printStackTrace();
 }
}
```

Fig. 6. The implementation of the method `bar` in the system S calling the method `foo` of the class `Foo` in I_E. We notice that in the argument vector of `foo` a fresh copy of the byte array is passed to the environment. Moreover, both the exception, possibly thrown by `foo`, and the returned object are immediately cloned and not used afterwards.

above. Moreover, the example illustrates the method `bar` in the system S calling the method `foo` in the environment E (see Figure 5). Each argument except from primitive values and strings is cloned before being passed to `foo` and both the returned object and the exception possibly thrown by the method are immediately cloned and not used afterwards.

We now state the main result of this paper: The following theorem is a generalisation of Theorem 4 to the richer family of programs considered in this section.

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

The proof of this theorem, as long as all the necessary lemmas to demonstrate it, are stated in Appendix A.2.

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

References

1. M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations Among Notions of Security for Public-Key Encryption Schemes. In H. Krawczyk, editor, *Advances in Cryptology, 18th Annual International Cryptology Conference (CRYPTO 1998)*, volume 1462 of *Lecture Notes in Computer Science*, pages 549–570. Springer, 1998.
2. R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. Technical Report 2000/067, Cryptology ePrint Archive, 2000. Available at `http://eprint.iacr.org/2000/067` with new versions from December 2005 and July 2013.
3. 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)*, pages 136–145. IEEE Computer Society, 2001.
4. Joseph A. Goguen and José Meseguer. Security Policies and Security Models. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
5. 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.
6. 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.
7. 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.
8. R. Küsters. Simulation-Based Security with Inexhaustible Interactive Turing Machines. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW-19 2006)*, pages 309–320. IEEE Computer Society, 2006. See `http://eprint.iacr.org/2013/025/` for a full and revised version.
9. R. Küsters, A. Datta, J. C. Mitchell, and A. Ramanathan. On the Relationships Between Notions of Simulation-Based Security. *Journal of Cryptology*, 21(4):492–546, 2008.

10. Ralf Küsters, Enrico Scapin, Tomasz Truderung, and Jürgen Graf. Extending and Applying a Framework for the Cryptographic Verification of Java Programs. In Martín Abadi and Steve Kremer, editors, *Principles of Security and Trust - Third International Conference, POST 2014*, volume 8414 of *Lecture Notes in Computer Science*, pages 220–239. Springer, 2014. A full version is available at http://eprint.iacr.org/2014/038.
11. Ralf Küsters, Tomasz Truderung, and Jürgen Graf. A Framework for the Cryptographic Verification of Java-like Programs. In *25th IEEE Computer Security Foundations Symposium (CSF 2012)*, pages 198–212. IEEE Computer Society, 2012.
12. 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)*, pages 270–284. IEEE Computer Society, 2008.
13. 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`.
14. Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing Robust Declassification and Qualified Robustness. *Journal of Computer Security*, 14(2):157–196, 2006.
15. Tobias Nipkow and David von Oheimb. Java$_{light}$ is Type-Safe — Definitely. In *POPL*, pages 161–170, 1998.
16. B. Pfitzmann and M. Waidner. A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission. In *IEEE Symposium on Security and Privacy*, pages 184–201. IEEE Computer Society, 2001.
17. Andrei Sabelfeld and Andrew C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications, special issue on Formal Methods for Security*, 21(1):5–19, 2003.
18. Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security, IOS Press*.
19. Andrei Sabelfeld and David Sands. Dimensions and Principles of Declassification. In *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005)*. IEEE Computer Society, 2005.

A A Proof Technique for Noninterference in Open Systems

A.1 Proof of Theorem 4

For completeness, we now repeat the proof of Theorem 4 as presented in the appendix of [11].

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

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

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

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

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

We can split the run ρ into segments

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

such that:

- Every A_i is a sequence of states (sub-run) where code of S is executed (formally transitions within A_k are labeled by names of classes from S). Moreover, every A_i except for the last one ends with a state of the form $(e_i[C_i.m_i(\vec{a}_i)], s_i)$ where the subexpression $C_i.m_i(\vec{a}_i)$ is about to be rewritten (with C_i defined in E). We will denote the tuple $(C_i, m_i, \vec{a}_i)$ by x_i.
- Every B_i is a sequence of states where code of E is executed. It begins with $(e_i[\{e'_i\}_{C_i}], s_i)$, where $\{e'_i\}_{C_i}$ is the block obtained by the static method call rule applied to $C_i.m_i(\vec{a}_i)$ (it depends only on C_i, m_i, and $\vec{a}_i$), and ends with $(e_i[\{y_i\}_{C_i}], t_i)$, where y_i is a value (that is return by this method).

We will represent such a run as

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

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

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

Lemma 1. *We have:*

1. $s_i|_S = t_i|_S$,
2. $t_i|_E = s_{i+1}|_E$.

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

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

Lemma 2. *Let S, S' and E be like in the theorem. Let ρ be the run of $E \cdot S$ and ρ' be the run of $E \cdot S'$. If*

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

then

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

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

Lemma 3. *Let S, E and E' be like in the theorem. Let ρ be the run of $E \cdot S$ and ρ' be the run of $E' \cdot S$. If*

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

then

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

Proof of Theorem 4. Implication from left to right is obvious. So let us assume that, I-non-interference *does not hold* for S. It means that there exists an I-environment E for S such that non-interference does not hold for $E \cdot S$. This, in turn, means that there are valid $\vec{a}_1$ and $\vec{a}_2$ such that the run ρ of $E \cdot S[a_1]$ and the run ρ' of $E \cdot S[a_1]$ give different results (i.e. both runs are finite and the final value of `result` is different).

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

As the value of `result` in a state s is part of $s|_E$, we conclude from Lemma 2, that there exists an index k such that $x_k^\rho \neq x_k^{\rho'}$. Let k be the first such index. Note that $y_i^\rho = y_i^{\rho'}$ for $i \in \{1, \ldots, k-1\}$. Let us assume that the first argument in the call described by x_k^ρ has value z which is different than the value z' of the first argument in $x_k^{\rho'}$ (for the other cases the proof is very similar).

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

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

– Let l be the integer such that u_l decides whether to test the first arguments in B_k-th block (then this argument is compared to u_{l+1} to determine the result). We set u_l to 1 and u_{l+1} to z (as defined above). Note that $l \notin \{p_1, \ldots, p_{k-1}\}$.
– As mentioned, for all remaining i we set $u_i = 0$.

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

Let σ be the run of the system $\tilde{E}_{\vec{u}} \cdot S[a_1]$ and σ' be the run of the system $\tilde{E}_{\vec{u}} \cdot S[a_2]$. As, by the definition of $\vec{u}$, we know that $y_i^{\rho} = y_i^{\rho'} = y_i^{\sigma} = y_i^{\sigma'}$ for $i \in \{1, \ldots, k-1\}$, we can use Lemma 3 to obtain $x_k^{\sigma} = x_k^{\rho}$ and $x_k^{\sigma'} = x_k^{\rho'}$. Therefore, the first argument in x_k^{σ} is z and it is different that the first argument in $x_k^{\sigma'}$. Therefore, by the definition of $\vec{u}$ (more precisely, by the values of u_l and u_{l+1} the variable `result` is set to `true` in $\tilde{E}_{\vec{u}} \cdot S[a_1]$ and to `false` in $\tilde{E}_{\vec{u}} \cdot S[a_2]$, after which both systems terminate (the `abort` is executed immediately after the assignment). Hence, these systems give different results, which completes the proof.

A.2 Proof of Theorem 5

The proof of Theorem 5 is quite similar to the proof of Theorem 4. The main difference is in the communication through strings, where indeed string references are exchanged, leading E and S to share part of their state. However, since strings are *immutable objects*, i.e., their internal state can not be modified after their creation, the part of the shared state of a system can not be modified by the other one. On the other hand side, the assumptions $E.1$-$E.3$ and $S.1$-$S.3$, respectively on the interface I_E and on the system S, guarantee that—even if, technically, some references are exchanged between E and S—the communication between them is, effectively, as if only pure values were exchanged.

We firstly (re)introduce the notation used in the previous section in order to *better* clarify it w.r.t the communication, besides through primitive types, also through strings, simple objects, arrays, and exception. Let I_E and S be as in the Theorem 5. Let $\emptyset \vdash E : I_E$ be an environment for S. We represent the run ρ of $E \cdot S$ as:

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

where the square brackets contain all the information that is passed between the two systems.

Thanks to the restrictions $S.1$ and $S.2$ regarding simple objects and arrays, we can assume that, other than to produce a fresh copy of them, E and S only share string references. We can then define the components of ρ in the following way:

– s_i is the state of the run at the end of the segment A_i, i.e., where the code of S is executed.
– $x_i = (C_i, m_i, \vec{a}_i)$ is a tuple denoting the call of the method m_i of the class C_i defined in E with arguments $\vec{a}_i$. The arguments vector $\vec{a}_i$ records primitive types, strings, *values* of arrays of primitive types (not the references of these arrays though), or *values* of simple objects (but, again, not their references), that is collections of all values of the fields of objects whose classes are defined in I_E.

- t_i is the state of the run at the end of the B_i segment, i.e., where the code of E is executed.
- y_i is the value returned by the method $C_i.m_i$ of E to S. Again, y_i may be a primitive value, a string, or a *value* either of an array (of primitive types), or of a simple object.

Furthermore, given a state s, let $s|_S$ be the part of s the system S can access and, similarly, let $s|_E$ be the part of s accessible by the environment E.

The following Lemma is an extension of a result, so called, state separation of E and S introduced in [11]: intuitively, for a representation of a run as above, B_i does not change the part of the state that can be reached from S and, similarly, A_i does not change the part of the state that can be reached from E.

Lemma 4. *Let S and E be two systems like in Theorem 5 and whose run ρ is defined as above. Then, for each $i \in \{1,\ldots,n-1\}$ we have:*

1. $s_i|_S = t_i|_S$,
2. $t_i|_E = s_{i+1}|_E$.

Proof. (Sketch) The first statement asserts that the part of the state accessible by S is, at the end of each segment A_i of the run, equal to the part of the state accessible by S at the end of the following segment B_i. Similarly, the second statement asserts that the part of the state accessible by E is, at the end of each segment B_i of the run, equal to the part of the state accessible by E at the end of the following segment A_{i+1}.

Altought formally simple objects, arrays, and exceptions are passed between the system S and the environment E, in Section 4.2 we imposed some restrictions which guarantee that the communication between S and E actually resembles exchange only of primitive values or string references. In particular, imposing that only fresh copies of references (different from strings) are passed to the enviorment and that every reference r (different from string) the environment returns to the system is immediately cloned and not used afterwards allows us to immediately conclude that E and S do not share none of this kind of references. Therefore, we can assume the systems E and S exchange only primitive values or string references.

Since in Jinja+ (as in Java) data is passed *by value*, both primitive values and string references are copied when they are exchanged between these systems. However, while if only primitive values are exchanged the part of the state accessible by E is disjoint to the part of the state accesible by S, in case strings are passed too, the system receiving their references is able to access also to a part of the state accessible from the other system. Nevertheless, since in Jinja+ (and in Java too) string objects are immutable and without any field, the system which receives their reference can not modify them. Therefore, in any case, at the end of each segment of the run, the part of the state accessible by the system whose code is going to be executed in the next segment has remained unchanged as it was at the beginning the last segment.

Because of this state separation, we can obtain two lemmas asserting that the part of the state accessible by each system and the values this system, at the end of each segment of its run, pass to the other one depend solely on the values exchanged between them so far.

Let us first introduce some notation and definitions. Let ρ and $\hat{\rho}$ be the runs of two systems, for example $E \cdot S$ and $E \cdot \hat{S}$. As already introduced before, $x_i/\hat{x}_i$ and $y_i/\hat{y}_i$ denote, respectively, the calls of the method m_i of the class C_i defined in E with arguments $\vec{a}_i/\vec{a}'_i$ and the values returned by them. We remember that they both record either primitive values or string references.

Let $f : \hat{R} \to R$ be a bijection from $\hat{R}$ to R, where $\hat{R}$ and R are subsets of the set of all string references, such that for each $\hat{r}$ and r such that $f(\hat{r}) = r$, their string values are the same. We denote it as $r \sim_f \hat{r}$. We extend the domain of f to expressions, states, configurations, and tuples $x_i/\hat{x}_i$. We do it, in the standard way, by structural isomorphism. That is, for example, $e \sim_f \hat{e}$ holds if and only if e and $\hat{e}$ are (syntactically) equal, up to references occurring as their corresponding subexpressions which need to be in the relation $\sim_f$, too. On the other hand side, for primitive values $v/\hat{v}$, we write $v \sim_f \hat{v}$, if simply $v = \hat{v}$.

Let $f\colon \hat{R}_f \to R_f$ and $f'\colon \hat{R}_{f'} \to R_{f'}$ be two congruences. We say that f' is *compatible* with f if for each $r \in \hat{R}_f \cap \hat{R}_{f'}$, we have $f(r) = f'(r)$. Furthermore, we say that f' is an *extension* of f (or, alternatively, f is a restriction of f') if f' is compatible with f, and, in particular, $\hat{R}_f \subseteq \hat{R}_{f'}$.

Lemma 5. *Let S_1, S_2 be two systems like the system S in Theorem 5 and let E be an I-environment for both S_1 and S_2. Let ρ be the run of $E \cdot S_1$ and $\hat{\rho}$ be the run of $E \cdot S_2$. If there exists a bijection $f\colon \hat{R}_f \to R_f$, where $\hat{R}_f$ and R_f denote the sets of all references in $\{\hat{x}_1, \ldots, \hat{x}_k, \hat{y}_1, \ldots, \hat{y}_{k-1}\}$ and in $\{x_1, \ldots, x_k, y_1, \ldots, y_{k-1}\}$ respectively, such that*

$$x_1 \sim_f \hat{x}_1, \ldots, x_k \sim_f \hat{x}_k \quad \textit{and} \quad y_1 \sim_f \hat{y}_1, \ldots, y_{k-1} \sim_f \hat{y}_{k-1},$$

then, there exists a bijection $f'\colon \hat{R}_{f'} \to R_{f'}$, where $\hat{R}_{f'}$ and $R_{f'}$ denote the sets of references in the domains of the heaps of $\hat{t}_k|_E$ and $t_k|_E$ respectively, such that

$$t_k|_E \sim_{f'} \hat{t}_k|_E \quad \textit{and} \quad y_k \sim_{f'} \hat{y}_k,$$

where f' is compatible with f.

Proof. By the premise of the lemma we know that there exists a bijection f such that for each $i \in \{1, \ldots, k\}$ we have $x_i \sim_f \hat{x}_i$ and for each $j \in \{1, \ldots, k-1\}$ we have $y_j \sim_f \hat{y}_j$. By the inductive hypothesis, we can assert that if there exists a bijection $g\colon \hat{R}_g \to R_g$ such that for each $i \in \{1, \ldots, k-1\}$ we have $x_i \sim_g \hat{x}_i$ and for each $j \in \{1, \ldots, k-2\}$ we have $y_j \sim_g \hat{y}_j$, then there exists a bijection $g'\colon \hat{R}_{g'} \to R_{g'}$, compatible with g, such that $t_{k-1}|_E \sim_{g'} \hat{t}_{k-1}|_E$ and $y_{k-1} \sim_{g'} \hat{y}_{k-1}$. We define the function g as a restriction of f, where $\hat{R}_g$ and R_g denote all references in $\{\hat{x}_1, \ldots, \hat{x}_{k-1}, \hat{y}_1, \ldots, \hat{y}_{k-2}\}$ and $\{x_1, \ldots, x_{k-1}, y_1, \ldots, y_{k-2}\}$, respectively. Then, we also have the function g' as above where $\hat{R}_{g'}$ and $R_{g'}$ denote the sets of references in the domains of the heaps of $\hat{t}_k|_E$ and $t_k|_E$, respectively. Moreover, as by Lemma 4 $t_{k-1}|_E = s_k|_E$ and $\hat{t}_{k-1}|_E = \hat{s}_k|_E$, we can assert $s_k|_E \sim_{g'} \hat{s}_k|_E$.

We want to show that g' is compatible with f. We will prove that for each $\hat{a}$ in $\hat{R}_{g'} \cap \hat{R}_f$, $\hat{a}$ is in $\hat{R}_g$ too. Then, since f is an extension of g, we have $f(\hat{a}) = g(\hat{a})$ and, since g is compatible with g', $f(\hat{a}) = g(\hat{a}) = g'(\hat{a})$, i.e., f and g' are compatible. The only references which may be in $\hat{R}_f \cap \hat{R}_{g'}$ but not in $\hat{R}_g$ are those that are in $\hat{x}_k$. Let $\hat{a}$ be

a reference of the method call denoted by $\hat{x}_k$ which is also in $\hat{R}_{g'}$. Then, by the defintion of $\hat{R}_{g'}$, $\hat{a}$ is in the domain of the heap of $\hat{t}_{k-1}|_E = \hat{s}_k|_E$. In particular, if $\hat{a}$ is $\hat{y}_{k-1}$, by the definition of f and g', we have $f(\hat{y}_{k-1}) = y_{k-1} = g'(\hat{y}_{k-1})$. Otherwise we notice that, as $\hat{a}$ is part of $\hat{x}_k$, $\hat{a}$ is also in the domain of the heap of $\hat{s}_k|_S$. Since all references in the domain of the heap of $\hat{s}_k$ were uniquely created in a (previous) point of the run $\hat{\rho}$ and since in the domains of the heap of $\hat{s}_k|_E$ and $\hat{s}_k|_S$ there are only those references that can be reached by E and S respectively, $\hat{a}$ must have been created by one of the two systems and then passed to the other one either in a previous method call $\hat{x}_1, \ldots, \hat{x}_{k-1}$ or in a value $\hat{y}_1, \ldots, \hat{y}_{k-2}$ previously returned. That is, by the definition of g, $\hat{a}$ is in $\hat{R}_g$ too.

The segments B_k and $\hat{B}_k$ start with $(e_z[\{e'_k\}_{C_k}], s_k)$ and $(\hat{e}_z[\{\hat{e}'_k\}_{C_k}], \hat{s}_k)$ respectively, where $\{e_k\}_{C_k}/\{\hat{e}'_k\}_{C_k}$ are the blocks obtained by the static method call rule (Rule 61 of Jinja+) applied to the call described by $x_k/\hat{x}_k$. In particular, $\{e'_k\}_{C_k} = \{V_1 : T_1, \ldots, V_n : T_n; V_1 := a_1, \ldots, V_n := a_n; \underline{e}\}_{C_k}$ and $\{\hat{e}'_k\}_{C_k} = \{V_1 : T_1, \ldots, V_n : T_n; V_1 := \hat{a}_1, \ldots, V_n := \hat{a}_n; \underline{e}\}_{C_k}$, where for each $j \in \{1, \ldots, l\}$ we have $a_j \sim_f \hat{a}_j$ and $\underline{e}$ is the body of the method m_k of E. Then, since f and g' are compatible, we can assert $(\{e'_k\}_{C_k}, s_k|_E) \sim_h (\{\hat{e}'_k\}_{C_k}, \hat{s}_k|_E)$ for a bijection $h : \{\hat{a}_1, \ldots, \hat{a}_n\} \cup \hat{R}_{g'} \to \{a_1, \ldots, a_n\} \cup R_{g'}$ defined in the following way: (i) $h(r) = f(r)$ if $r \in \{\hat{a}_1, \ldots, \hat{a}_n\}$; (ii) $h(r) = g'(r)$ if $r \in \hat{R}_{g'} \setminus \{\hat{a}_1, \ldots, \hat{a}_n\}$. That is, h maps references in $\hat{x}_k$ to references in x_k and references in the heap of $\hat{s}_k|_E$ to references in the heap of $s_k|_E$. We notice that h is compatible with both f and g', and therefore also with g.

As applicability of no Jinja+ rule depends on the particular value of references and as the blocks $\{e'_k\}_{C_k}/\{\hat{e}'_k\}_{C_k}$ are syntactically the same up to the references in their initial assignments, the rules applied in these blocks depend solely on $\underline{e}$. Let

$$(e_0, u_0) \to (e_1, u_1) \to \cdots \to (e_l, u_l),$$

be the segment B_k where its expressions are cut off from the set of classes belonging to S and where only the part of the state that E can modify is taken into account. In particular, $(e_0, u_0) = (\{e'_k\}_{C_k}, s_k|_E)$ and $(e_l, u_l) = (\{y_k\}_{C_k}, t_k|_E)$. Let

$$(\hat{e}_0, \hat{u}_0) \to (\hat{e}_1, \hat{u}_1) \to \cdots \to (\hat{e}_l, \hat{u}_l),$$

be the segment $\hat{B}_k$ pruned in the same way as B_k, where $(\hat{e}_0, \hat{u}_0) = (\{\hat{e}'_k\}_{C_k}, \hat{s}_k|_E)$ and $(\hat{e}_l, \hat{u}_l) = (\{\hat{y}_k\}_{C_k}, \hat{t}_k|_E)$.

We show that for each pair of configurations $q_j/\hat{q}_j$ in the two (sub-)runs defined above, there exists a bijection $h_j : \hat{R}_j \to R_j$ such that $q_i = (e_j, s_j) \sim_{h_j} (\hat{e}_j, \hat{s}_j) = \hat{q}_j$ and such that h_j is a extension of the bijection h_{j-1} among (e_{j-1}, s_{j-1}) and $(\hat{e}_{j-1}, \hat{s}_{j-1})$. We do it by induction on the number of steps of execution.

`Base case`: $j = 0$. As we know $(\{e'_k\}_{C_k}, s_k|_E) \sim_h (\{\hat{e}'_k\}_{C_k}, \hat{s}_k|_E)$, we have $h_0 = h$.

`Inductive Step`: Let us assume, by the inductive hypothesis, that there exists a bijection $h_j : \hat{R}_j \to R_j$ for $0 < j < n$ such that $(e_j, u_j) \sim_{h_j} (\hat{e}_j, \hat{u}_j)$ and such that h_j is an extension of h_{j-1}. Since $(e_j, u_j) \to (e_{j+1}, u_{j+1})$ and $(\hat{e}_j, \hat{u}_j) \to (\hat{e}_{j+1}, \hat{u}_{j+1})$, we prove that there exists a bijection $h_{j+1} : \hat{R}_{j+1} \to R_{j+1}$ such that $(e_{j+1}, u_{j+1}) \sim_{h_{j+1}} (\hat{e}_{j+1}, \hat{u}_{j+1})$ and $\hat{R}_j \subseteq \hat{R}_{j+1}$. We distinguish four cases depending on the behavior of the Jinja+ rule applied in the j-th step of computation:

a) The effect of the rule on the state is directly inherited from the reduction step of a subexpression in the hypotheses the rule (this case holds for all the subexpression

reduction rules of Jinja+). In this case, there exist two subexpressions e and $\hat{e}$ of the expressions e_j and $\hat{e}_j$ respectively, whose reduction step directly determines the states $u_{j+1}/\hat{u}_{j+1}$. Since h_j is defined by structural isomorphism, (e,u_j) and $(\hat{e},\hat{u}_j)$ are under the bijection h_j too. By the inductive hyphothesis, the configurations resulting from the reduction step of the subexpressions $e/\hat{e}$ are under a bijection h_{j+1} which extends h_j. Hence, since h_{j+1} is also defined by structural isomorphism, we can assert that the configurations (e_{j+1},u_{j+1}) and $(\hat{e}_{j+1},\hat{u}_{j+1})$ are under the bijection h_{j+1}, too.

b) The rule creates a new reference in the state of computations (Rules 16, 62, 71, 72, 76 of Jinja+). In this case, in (e_{j+1},u_{j+1}) and $(\hat{e}_{j+1},\hat{u}_{j+1})$ occur two fresh references a and $\hat{a}$ unused in u_j and $\hat{u}_j$, respectively. Since, by the inductive hypothesis, h_j is an extension of $h_0 = h$, and, by the definition of h, $\mathtt{dom}(h)$ is the union of references in $\hat{x}_k$ with references in the heap of $\hat{u}_0 = \hat{s}_k|_E$, then $\mathtt{dom}(h_j)$ differs from $\mathtt{dom}(h_0)$ solely in the set of references which have been created in the segment $\hat{B}_k$ so far, i.e., which are in the heap of $\hat{u}_j$ but not in the heap of $\hat{u}_0$. Therefore, if $\hat{a}$ is not in the heap of $\hat{u}_j$, then it is not even in $\mathtt{dom}(h_j) = \hat{R}_j$. We can then extend the bijection h_j to another bijection h_{j+1} in the following way: (i) $\hat{R}_{j+1} = \hat{R}_j \cup \{\hat{a}\}$; (ii) $\forall r \in \hat{R}_j, h_j(r) = h_{j+1}(r)$; (iii) $a = h_{j+1}(\hat{a})$. Since all the other references remain unchanged, we can conclude $(e_{j+1},u_{j+1}) \sim_{h_{j+1}} (\hat{e}_{j+1},\hat{u}_{j+1})$.

c) The rules update an entry in the state of computations (Rule 20 of Jinja+ performs an update on the store, whereas Rules 23 and 68 perform an update on the heap). In these (three) rules, the expressions e_j and $\hat{e}_j$ are reduced to e_{j+1} and $\hat{e}_{j+1}$ by trimming the updating value off. Therefore, since no new references occur in $e_{j+1}/\hat{e}_{j+1}$, we have $e_{j+1} \sim_{h_j} \hat{e}_{j+1}$. Furthermore, the states $u_j/\hat{u}_j$ are updated with values which are in $e_j/\hat{e}_j$ and hence, in case these values are references, already under the bijection h_j. Therefore, the entries updated remain under the same bijection h_j, by which we can conclude $(e_{j+1},u_{j+1}) \sim_{h_{j+1}} (\hat{e}_{j+1},\hat{u}_{j+1})$, where $h_{j+1} = h_j$.

d) The rule leaves the state of the configurations unchanged: this case holds for all the expression reduction rules and all the exception propagation rules of Jinja+ not mentioned in the previous two items. Here, if in $e_{j+1}/\hat{e}_{j+1}$ occur new references, they must have been in $u_j/\hat{u}_j$ and therefore under the bijection h_j. (This case could for example happen in the field access rule, Rule 22, if the field is not of primitive type.) We can then assert $e_{j+1} \sim_{h_j} \hat{e}_{j+1}$. Moreover, since their states remain unchanged, we can conclude $(e_{j+1},u_{j+1}) \sim_{h_{j+1}} (\hat{e}_{j+1},\hat{u}_{j+1})$, where $h_{j+1} = h_j$.

Therefore, since in each step j of these two runs the configurations are under a bijection h_j extending the initial bijection $h_0 = h$, at the end of the segments $B_k/\hat{B}_k$ there exists a bijection h_l, which extends h, such that $(\{y_k\}_{C_k}, t_k|_E) \sim_{h_l} (\{\hat{y}_k\}_{C_k}, \hat{t}_k|_E)$. We can then set $f' = h_l$ to obtain $t_k|_E \sim_{f'} \hat{t}_k|_E$ and $y_k \sim_{f'} \hat{y}_k$.

In order to conclude the proof, we have to show that f' is compatible with f. The bijection f' is an extension of h which is compatible with f. That is, f' is also compatible with f.

Lemma 6. *Let S be like in Theorem 5 and let E_1 and E_2 be two I-environments for S. Let ρ be the run of $E_1 \cdot S$ and $\hat{\rho}$ be the run of $E_2 \cdot S$. If there exists a bijection $f\colon \hat{R}_f \to R_f$, where $\hat{R}_f$ and R_f denote the sets of all references in $\{\hat{x}_1,\ldots,\hat{x}_k,\hat{y}_1,\ldots,\hat{y}_k\}$ and in*

$\{x_1,\ldots,x_k,y_1,\ldots,y_k\}$ *respectively, such that*

$$x_1 \sim_f \hat{x}_1,\ldots,x_k \sim_f \hat{x}_k \quad \textit{and} \quad y_1 \sim_f \hat{y}_1,\ldots,y_k \sim_f \hat{y}_k,$$

then, there exists a bijection $f' : \hat{R}_{f'} \to R_{f'}$*, where* $\hat{R}_{f'}$ *and* $R_{f'}$ *denote the sets of references in the domains of the heaps of* $\hat{s}_{k+1}|_S$ *and* $s_{k+1}|_S$ *respectively, such that*

$$s_{k+1}|_S \sim_{f'} \hat{s}_{k+1}|_S \quad \textit{and} \quad x_{k+1} \sim_{f'} \hat{x}_{k+1},$$

where f' *is compatible with* f*.*

Proof. (Sketch) The proof of this Lemma is somewhat complementary to the proof of Lemma 5: the segments $A_{k+1}/\hat{A}_{k+1}$ we are considering now are two sequences of n configurations of S pairwise equal, up to those references which have been exchanged with E_1 and E_2 so far and which are, by the hypothesis of the lemma, under the bijection f.

By following the same inductive reasoning as in the proof of the other lemma, there exist two bijections g and g' such that, respectively, g is defined as a restriction of f to the domain containing all references in $\{\hat{x}_1,\ldots,\hat{x}_{k+1},\hat{y}_1,\ldots,\hat{y}_{k+1}\}$ and g' maps references in the domain of the heap of $\hat{s}_k|_S = \hat{t}_k|_S$ to references in the domain of the heap of $s_k|_S = t_k|_E$. Therefore, we can assert that the two initial configurations $(e_z,t_k)/(\hat{e}_z,\hat{t}_k)$ of the segments $A_{k+1}/\hat{A}_{k+1}$, pruned from the part of the state not accessible from S, are under a bijection h defined, as in the other lemma, piecewise by f and g'. (Bijections f and g' are compatible since all references in $\hat{R}_f \cap \hat{R}_{g'}$ are in the domain of g too; see the corresponding reasoning in the proof of Lemma 5 for details.)

As applicability of no Jinja+ rule depends on the particular value of references, the same rule is applied in each step of A_{k+1} and $\hat{A}_{k+1}$. Therefore, by induction on the number of steps we can assert that, at the end of the two segments, there exists a bijection f', compatible with f, among their final (pruned) configurations, $(e_{z+n}[C_{k+1}.m_{k+1}(\vec{a}_{k+1})],s_{k+1}|_S)$ and $(\hat{e}_{z+n}[C_{k+1}.m_{k+1}(\vec{a}'_{k+1})],\hat{s}_{k+1}|_S)$, where $(C_{k+1},m_{k+1},\vec{a}_{k+1}) = x_{k+1}$ and $(C_{k+1},m_{k+1},\vec{a}'_{k+1}) = \hat{x}_{k+1}$, respectively. (Again, see the final part of the proof of Lemma 5 for details.)

We can now prove the main theorem of our extension. Thanks to the assumptions on the components of the run ρ, we remind that only primitives values and string references are exchanged between S and E.

Proof of Theorem 5. Implication from left to right is obvious. So let us assume that, I-non-interference *does not hold* for S. It means that there exists an I-environment E for S such that non-interference does not hold for $E \cdot S$, i.e., there are valid $\vec{a}_1$ and $\vec{a}_2$ such that the run ρ of $E \cdot S[\vec{a}_1]$ and the run $\hat{\rho}$ of $E \cdot S[\vec{a}_2]$ give different results.

We can define a sequence $\vec{u}$ such that the system $\tilde{E}_{\vec{u}}$ behaves exactly like E and it is therefore able to distinguish between $S[\vec{a}_1]$ and $S[\vec{a}_2]$. Since the value of `result` in a state s is part of $s|_E$, we conclude from Lemma 5, that there exists an index k such that x_k^ρ and $x_k^{\hat{\rho}}$ are not under a bijection. Let k be the first such index. Note that, from Lemma 5, we also know $x_i^\rho \sim_f x_i^{\hat{\rho}}$ and $y_i^\rho \sim_f y_i^{\hat{\rho}}$ for $i \in \{1,\ldots,k-1\}$ and for a bijection f. Let us assume that what breaks the bijection are the first arguments z^ρ and $z^{\hat{\rho}}$ in the calls described by x_k^ρ and $x_k^{\hat{\rho}}$, respectively (for the other cases the proof is very similar).

We can now define the sequence $\vec{u}$ which determines the behavior of $\tilde{E}_{\vec{u}}$ as a sequence containing only zeros with the following exceptions:

- For each primitive value v_i that E returns to $S[\vec{a}_1]$, let l be the index such that u_l is the element of $\vec{u}$ determining the i-th primitive value that $\tilde{E}_{\vec{u}}$ returns to $S[\vec{a}_1]$. We then set $u_l = v_i$.
- Each string reference r_i that E returns to $S[\vec{a}_1]$ can be either a reference already exchanged between E and $S[\vec{a}_1]$ before or a freshly exchanged one. In the system $\tilde{E}_{\vec{u}} \cdot S[\vec{a}_1]$, let p be the integer such that u_p decides whether to return a freshly created string or a string stored in `stringList`.
  If r_i is a freshly exchanged reference, we set u_p to 1, u_{p+1} to the length of r_i and each u_j for $j \in \{p+2, \ldots, p+|r_i|+1\}$ to the integer corresponding to the $(j-p-2)$-th character of r_i. Otherwise, if r_i was already exchanged before the last segment of execution of E, let assume this reference is the j-th element of `stringList`. We set u_p to 2 and u_{p+j} to 1.
- The arguments z^{ρ} and $z^{\hat{\rho}}$ brake the bijection either because their values are different or because they point to different references in $s|_E$. In the system $\tilde{E}_{\vec{u}} \cdot S[\vec{a}_1]$, if z^{ρ} and $z^{\hat{\rho}}$ are primitive values, then let q be the integer such that u_q decides whether to test these arguments (they are then compared to u_{q+1} to determine the result). We set $u_q = 1$ and $u_{q+1} = z^{\rho}$.
  On the other hand side, if z^{ρ} and $z^{\hat{\rho}}$ are string references, we distinguish the following three cases: (1) If the two string values have different lengths, let l be the integer such that u_l decides whether to test their lengths. We then set u_l to 1 and u_{l+1} to the length of z^{ρ}. (2) Otherwise, if the two string values differ from at least one character, let p be the integer such that u_p decides whether to test the j-th character of z^{ρ}, the first one which differs from the j-th character of $z^{\hat{\rho}}$. We set u_p to 1 and u_{p+1} to the integer corresponding to the j-th character of z^{ρ}. (3) Finally, if there exists (at least) an element of `stringList` whose reference comparison with z^{ρ} gives a different result to the reference comparison with $z^{\hat{\rho}}$, let q be the integer such that u_q decides whether to test the result of the comparison between $z^{\rho}/z^{\hat{\rho}}$ and the aforementioned element of `stringList`. We set both u_q and u_{q+1} to 1.
- As mentioned, for all remaining i we set $u_i = 0$.

In order to complete the proof, it is enough to show that $\tilde{E}_{\vec{u}} \cdot S[\vec{a}_1]$ and $\tilde{E}_{\vec{u}} \cdot S[\vec{a}_2]$ give different results.

Let σ be the run of the system $\tilde{E}_{\vec{u}} \cdot S[\vec{a}_1]$ and $\hat{\sigma}$ be the run of the system $\tilde{E}_{\vec{u}} \cdot S[\vec{a}_2]$. Therefore, by the definition of $\vec{u}$, we have that there exist two bijections g and h such that $x_i^{\rho} \sim_g x_i^{\sigma}$, $y_i^{\rho} \sim_g y_i^{\sigma}$ and $x_i^{\hat{\rho}} \sim_h x_i^{\hat{\sigma}}$, $y_i^{\hat{\rho}} \sim_h y_i^{\hat{\sigma}}$ for $i \in \{1, \ldots, k-1\}$. Then, by Lemma 6, there exist two bijections g' and h', compatible with g and h respectively, such that $x_k^{\rho} \sim_{g'} x_k^{\sigma}$ and $x_k^{\hat{\rho}} \sim_{h'} x_k^{\hat{\sigma}}$: In particular, $z^{\rho} = g'(z^{\sigma})$ and $z^{\hat{\rho}} = h'(z^{\hat{\sigma}})$. Since, by the construction of $\tilde{E}_{\vec{u}}$, we know that there not exists a bijection f such that $z^{\rho} = f(z^{\hat{\rho}})$, then there not exists neither a bijection f' such that $z^{\sigma} = f'(z^{\hat{\sigma}})$. In fact, if the bijection f' existed, it should be defined as $f' = (g')^{-1} \circ f \circ h'$, i.e., the composition of the inverse of g' with (the nonexistent) f, composed with h'. We can then conclude that z^{σ} and $z^{\hat{\sigma}}$ are not under a bijection either. Therefore, by the definition of $\vec{u}$, the variable `result` is set to `true` in $\tilde{E}_{\vec{u}} \cdot S[\vec{a}_1]$ and to `false` in $\tilde{E}_{\vec{u}} \cdot S[\vec{a}_2]$, after which both systems terminate

(the `abort` is executed immediately after the assignment). Hence, these systems give different results, which completes the proof.

B Rules of Jinja+

In this section we summarize all the rules of Jinja+. We start with rules of Jinja, following [7] (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).

There are two points where our presentation rules diverge from the ones of [7]. First, as we assume *unbounded memory*, we do not have rules which throw `OutOfMemoryError` (and we assume that $(new\text{-}Addr\ h)$ 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 (8)–(10)).

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 7) 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 8) 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*:

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

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

Exceptional reduction and *exception propagation* rules (Figure 9 and 10) 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.

B.1 Rules of Jinja+

In this section we present additional rules of Jinja+. Theres rules concern static method invocation, arrays, and strings. The rules are given in Figure 11, 12, and 13.

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle \texttt{Cast}\ C\ e,s\rangle \xrightarrow{\ell} \langle \texttt{Cast}\ C\ e',s'\rangle} \tag{1}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle V := e,s\rangle \xrightarrow{\ell} \langle V := e',s'\rangle} \tag{2}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle e.F\{D\},s\rangle \xrightarrow{\ell} \langle e'.F\{D\},s'\rangle} \tag{3}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle e.F\{D\} := e_2,s\rangle \xrightarrow{\ell} \langle e'.F\{D\} := e_2,s'\rangle} \tag{4}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle \texttt{Val}\ v.F\{D\} := e,s\rangle \xrightarrow{\ell} \langle \texttt{Val}\ v.F\{D\} := e',s'\rangle} \tag{5}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle e \ll bop \gg e_2,s\rangle \xrightarrow{\ell} \langle e' \ll bop \gg e_2,s'\rangle} \tag{6}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle \texttt{Val}\ v_1 \ll bop \gg e,s\rangle \xrightarrow{\ell} \langle \texttt{Val}\ v_1 \ll bop \gg e',s'\rangle} \tag{7}$$

$$\frac{P \vdash \langle e,(h,l(V := None))\rangle \xrightarrow{\ell} \langle e',(h',l')\rangle \quad l'\ V = None \quad \neg\ assigned\ V\ e}{P \vdash \langle \{V : T; e\}_D,(h,l)\rangle \xrightarrow{f(l,D)} \langle \{V : T; e'\}_D,(h',l'(V := l\ V))\rangle} \tag{8}$$

$$\frac{P \vdash \langle e,(h,l(V := None))\rangle \xrightarrow{\ell} \langle e',(h',l')\rangle \quad l'\ V = v \quad \neg\ assigned\ V\ e}{P \vdash \langle \{V : T; e\}_D,(h,l)\rangle \xrightarrow{f(l,D)} \langle \{V : T;\ V := \texttt{Val}\ v; e'\}_D,(h',l'(V := l\ V))\rangle} \tag{9}$$

$$\frac{P \vdash \langle e,(h,l(V := v))\rangle \xrightarrow{\ell} \langle e',(h',l')\rangle \quad l'\ V = v'}{P \vdash \langle \{V : T; V := \texttt{Val}\ v; e\}_D,(h,l)\rangle \xrightarrow{f(l,D)} \langle \{V : T;\ V := \texttt{Val}\ v'; e'\}_D,(h',l'(V := l\ V))\rangle} \tag{10}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle e.M(es),s\rangle \xrightarrow{\ell} \langle e'.M(es),s'\rangle} \tag{11}$$

$$\frac{P \vdash \langle es,s\rangle\ [\xrightarrow{\ell}]\ \langle es',s'\rangle}{P \vdash \langle \texttt{Val}\ v.M(es),s\rangle \xrightarrow{\ell} \langle \texttt{Val}\ v.M(es'),s'\rangle} \tag{12}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle e;e_2,s\rangle \xrightarrow{\ell} \langle e';e_2,s'\rangle} \tag{13}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle \texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2,s\rangle \to \langle \texttt{if}\ (e')\ e_1\ \texttt{else}\ e_2,s'\rangle} \tag{14}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{\ell} \langle e',s'\rangle}{P \vdash \langle e \cdot es,s\rangle\ [\xrightarrow{\ell}]\ \langle e' \cdot es,s'\rangle} \qquad \frac{P \vdash \langle es,s\rangle\ [\xrightarrow{\ell}]\ \langle es',s'\rangle}{P \vdash \langle \texttt{Val}\ v \cdot es,s\rangle\ [\xrightarrow{\ell}]\ \langle \texttt{Val}\ v \cdot es',s'\rangle} \tag{15}$$

Fig. 7. Subexpression reduction rules. We define $g(\ell,D) = D$, if $\ell = -$; otherwise $g(\ell,D) = \ell$.

$$\frac{\textit{new-Addr}\ h = a \quad P \vdash C\ \textit{has-fields}\ FDTs}{P \vdash \langle \texttt{new}\ C, (h,l)\rangle \xrightarrow{-} \langle \textit{addr}\ a, (h(a \mapsto\ (C,\ \textit{init-fields}\ FDTs)), l)\rangle} \tag{16}$$

$$\frac{hp\ s\ a = (D, fs) \quad P \vdash D \preceq^*\ C}{P \vdash \langle \texttt{Cast}\ C\ (\textit{addr}\ a), s\rangle \xrightarrow{-} \langle \textit{addr}\ a, s\rangle} \tag{17}$$

$$P \vdash \langle \texttt{Cast}\ C\ \textit{null}, s\rangle \xrightarrow{-} \langle \textit{null}, s\rangle \tag{18}$$

$$\frac{lcl\ s\ V = v}{P \vdash \langle \texttt{Var}\ V, s\rangle \xrightarrow{-} \langle \texttt{Val}\ v, s\rangle} \tag{19}$$

$$P \vdash \langle V := \texttt{Val}\ v, (h,l)\rangle \xrightarrow{-} \langle \textit{unit}, (h, l(V \mapsto\ v))\rangle \tag{20}$$

$$\frac{\textit{binop}\ (bop, v_1, v_2) = v}{P \vdash \langle \texttt{Val}\ v_1 \ll bop \gg \texttt{Val}\ v_2, s\rangle \xrightarrow{-} \langle \texttt{Val}\ v, s\rangle} \tag{21}$$

$$\frac{hp\ s\ a = (C, fs) \quad fs(F,D) = v}{P \vdash \langle \textit{addr}\ a.F\{D\}, s\rangle \xrightarrow{-} \langle \texttt{Val}\ v, s\rangle} \tag{22}$$

$$\frac{hp\ a = (C, fs)}{P \vdash \langle \textit{addr}\ a.F\{D\} := \texttt{Val}\ v, (h,l)\rangle \xrightarrow{-} \langle \textit{unit}, (h(a\ \mapsto\ (C, fs((F,D)\ \mapsto\ v))), l)\rangle} \tag{23}$$

$$\frac{hp\ s\ a = (C, fs) \quad P \vdash\ C\ \textit{sees}\ M:\ Ts \to T = (pns,\ body)\ \textit{in}\ D \quad |vs| = |pns| \quad |Ts| = |pns|}{P \vdash \langle \textit{addr}\ a.M(map\ \texttt{Val}\ vs), s\rangle \xrightarrow{-} \langle blocks_D(this \cdot pns,\ Class\ D \cdot Ts,\ \textit{addr}\ a \cdot vs,\ body), s\rangle} \tag{24}$$

$$P \vdash \langle \{V : T;\ \texttt{Val}\ u\}_D, s\rangle \xrightarrow{D} \langle \texttt{Val}\ u, s\rangle \tag{25}$$

$$P \vdash \langle \{V : T;\ V := \texttt{Val}\ v;\ \texttt{Val}\ u\}_D, s\rangle \xrightarrow{D} \langle \texttt{Val}\ u, s\rangle \tag{26}$$

$$P \vdash \langle \texttt{Val}\ v;\ e_2, s\rangle \xrightarrow{-} \langle e_2, s\rangle \tag{27}$$

$$P \vdash \langle \texttt{if}(\textit{true})\ e_1\ \texttt{else}\ e_2, s\rangle \xrightarrow{-} \langle e_1, s\rangle \tag{28}$$

$$P \vdash \langle \texttt{if}(\textit{false})\ e_1\ \texttt{else}\ e_2, s\rangle \xrightarrow{-} \langle e_2, s\rangle \tag{29}$$

$$P \vdash \langle \texttt{while}(b)\ c, s\rangle \xrightarrow{-} \langle \texttt{if}(b)\ (c;\ \texttt{while}(b)\ c)\ \texttt{else}\ \textit{unit}, s\rangle \tag{30}$$

Fig. 8. Expression reduction

$$\frac{hp\ s\ a\ =\ (D, fs) \quad \neg\, P \vdash D \preceq^* C}{P \vdash \langle \texttt{Cast}\ C(addr\ a), s\rangle \xrightarrow{-} \langle \texttt{THROW ClassCastException},\ s\rangle} \tag{31}$$

$$P \vdash \langle null.F\{D\}, s\rangle \xrightarrow{-} \langle \texttt{THROW NullPointerException},\ s\rangle \tag{32}$$

$$P \vdash \langle null.F\{D\} := \texttt{Val}\ v, s\rangle \xrightarrow{-} \langle \texttt{THROW NullPointerException},\ s\rangle \tag{33}$$

$$P \vdash \langle null.M(map\ \texttt{Val}\ vs), s\rangle \xrightarrow{-} \langle \texttt{THROW NullPointerException},\ s\rangle \tag{34}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle \texttt{throw}\ e, s\rangle \xrightarrow{\ell} \langle \texttt{throw}\ e', s'\rangle} \tag{35}$$

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

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle \texttt{try}\ e\ \texttt{catch}\ (C\ V)\ e_2, s\rangle \xrightarrow{\ell} \langle \texttt{try}\ e'\ \texttt{catch}\ (C\ V)\ e_2, s'\rangle} \tag{37}$$

$$P \vdash \langle \texttt{try}\ \ \texttt{Val}\ v\ \texttt{catch}\ (C\ V)\ e_2, s\rangle \xrightarrow{-} \langle \texttt{Val}\ v, s\rangle \tag{38}$$

$$\frac{hp\ s\ a\ =\ (D, fs) \quad P \vdash\ D\ \preceq^*\ C}{P \vdash \langle \texttt{try Throw}\ a\ \texttt{catch}\ (C\ V)\ e_2, s\rangle \xrightarrow{-} \langle \{V : Class\ C;\ V := addr\ a;\ e_2\}, s\rangle} \tag{39}$$

$$\frac{hp\ s\ a\ =\ (D, fs) \quad \neg\, P \vdash\ D\ \preceq^*\ C}{P \vdash \langle \texttt{try Throw}\ a\ \texttt{catch}\ (C\ V)\ e_2, s\rangle \xrightarrow{-} \langle \texttt{Throw}\ a, s\rangle} \tag{40}$$

Fig. 9. Exceptional expression reduction

$$P \vdash \langle \texttt{Cast}\ C\ (\texttt{throw}\ e), s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{41}$$

$$P \vdash \langle V := \texttt{throw}\ e, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{42}$$

$$P \vdash \langle \texttt{throw}\ e.F\{D\}, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{43}$$

$$P \vdash \langle \texttt{throw}\ e.F\{D\} := e_2, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{44}$$

$$P \vdash \langle \texttt{Val}\ v.F\{D\} := \texttt{throw}\ e, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{45}$$

$$P \vdash \langle \texttt{throw}\ e\ \ll bop \gg\ e_2, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{46}$$

$$P \vdash \langle \texttt{Val}\ v_1\ \ll bop \gg\ \texttt{throw}\ e, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{47}$$

$$P \vdash \langle \{V : T;\ \texttt{Throw}\ a\}_D, s\rangle \xrightarrow{D} \langle \texttt{Throw}\ a, s\rangle \tag{48}$$

$$P \vdash \langle \{V : T; V := \texttt{Val}\ v;\ \texttt{Throw}\ a\}_D, s\rangle \xrightarrow{D} \langle \texttt{Throw}\ a, s\rangle \tag{49}$$

$$P \vdash \langle \texttt{throw}\ e.M(es), s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{50}$$

$$P \vdash \langle \texttt{Val}\ v.M(map\ \texttt{Val}\ vs\ @\ (\texttt{throw}\ e\ \cdot\ es')), s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{51}$$

$$P \vdash \langle \texttt{throw}\ e;\ e_2, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{52}$$

$$P \vdash \langle \texttt{if}(\texttt{throw}\ e)\ e_1\ \texttt{else}\ e_2, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{53}$$

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

Fig. 10. Exception propagation

$$\frac{P \vdash \langle es, s\rangle \ [\xrightarrow{\ell}] \ \langle es', s'\rangle}{P \vdash \langle D.M(es), s\rangle \xrightarrow{\ell} \langle D.M(es'), s'\rangle} \tag{55}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle e[e_2], s\rangle \xrightarrow{\ell} \langle e'[e_2], s'\rangle} \tag{56}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle (\texttt{Val}\ v)[e], s\rangle \xrightarrow{\ell} \langle (\texttt{Val}\ v)[e'], s'\rangle} \tag{57}$$

$$P \vdash \langle D.M(map\ \texttt{Val}\ vs\ @\ (\texttt{throw}\ e\ \cdot\ es')), s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{58}$$

$$P \vdash \langle (\texttt{throw}\ e)[e'], s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{59}$$

$$P \vdash \langle e'[\texttt{throw}\ e], s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{60}$$

Fig. 11. Subexpression reduction and exception propagation rules for Jinja+.

$$\frac{P \vdash\ D \textit{ has-static } M :\ Ts \to T = (pns,\ body) \qquad |vs| = |pns| \qquad |Ts| = |pns|}{P \vdash \langle D.M(map\ \texttt{Val}\ vs), s\rangle \xrightarrow{-} \langle blocks_D(pns, Ts, vs,\ body), s\rangle} \tag{61}$$

$$\frac{n \geq 0,\ new\text{-}Addr\ h = a}{P \vdash \langle \texttt{new}\ \ \tau\texttt{[}intg(n)\texttt{]},\ (h,l)\rangle \xrightarrow{-} \langle addr\ a, (h(a \mapsto initArr(\tau, n)), l)\rangle} \tag{62}$$

$$P \vdash \langle null[intg(n)], s\rangle \xrightarrow{-} \langle \texttt{THROW NullPointerException},\ s\rangle \tag{63}$$

$$\frac{n < 0}{P \vdash \langle \texttt{new}\ \ \tau\texttt{[}intg(n)\texttt{]},\ (h,l)\rangle \xrightarrow{-} \langle \texttt{THROW NegativeArraySizeException},\ (h,l)\rangle} \tag{64}$$

$$\frac{h\ a = (\tau, m, t),\ 0 \leq n < m,\ h\ t(n) = v}{P \vdash \langle (addr\ a)\texttt{[}intg\ n\texttt{]},\ (h,l)\rangle \xrightarrow{-} \langle Val\ v,\ (h,l)\rangle} \tag{65}$$

$$\frac{h\ a = (\tau, m, t),\ \neg(0 \leq n < m),}{P \vdash \langle (addr\ a)\texttt{[}intg\ n\texttt{]},\ (h,l)\rangle \xrightarrow{-} \langle \texttt{THROW IndexOutOfBoundsException},\ (h,l)\rangle} \tag{66}$$

$$\frac{h\ a = (\tau, m, t),}{P \vdash \langle (addr\ a)\texttt{.lenght},\ (h,l)\rangle \xrightarrow{-} \langle intg\ m,\ (h,l)\rangle} \tag{67}$$

$$\frac{h\ a = (\tau, m, t),\ 0 \leq n < m,\ isOfType(v, \tau),\ t' = arrayUpdate(t, n, v)}{P \vdash \langle (addr\ a)\texttt{[}intg\ n\texttt{]} := Val\ v,\ (h,l)\rangle \xrightarrow{-} \langle unit,\ (h(a \mapsto (\tau, m, t')), l)\rangle} \tag{68}$$

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

$$\frac{h\ a = (\tau, m, t),\ 0 \leq n < m,\ \neg isOfType(v, \tau),}{P \vdash \langle (addr\ a)\texttt{[}intg\ n\texttt{]} := Val\ v,\ (h,l)\rangle \xrightarrow{-} \langle \texttt{THROW ArrayStoreException},\ (h,l)\rangle} \tag{70}$$

Fig. 12. (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 $body$.

$$\frac{\textit{new-Addr } h = a}{P \vdash \langle \textit{litS str},\ \langle h,l\rangle\rangle \xrightarrow{-} \langle \textit{addr } a, \langle h(a \mapsto \textit{initString}(\textit{str})), l\rangle\rangle} \tag{71}$$

$$\frac{h(a_1) = (ch_1, m_1),\ h(a_2) = (ch_2, m_2),\ \textit{new-Addr } h = a_3,\ ch_3 = \textit{concat}(ch_1, ch_2)}{P \vdash \langle (\textit{addr } a_1) + (\textit{addr } a_2), \langle h,l\rangle \xrightarrow{-} \langle (\textit{addr } a_3), \langle h(a_3 \mapsto (ch_3, m_1 + m_2)), l\rangle\rangle\rangle} \tag{72}$$

$$\frac{h(a) = (ch,m)}{P \vdash \langle (\textit{addr } a)\texttt{.lenght()},\ \langle h,l\rangle\rangle \xrightarrow{-} \langle \textit{intg } m, \langle h,l\rangle\rangle} \tag{73}$$

$$\frac{h(a) = (ch,m),\ 0 \leq n < m,\ ch[n] = c}{P \vdash \langle (\textit{addr } a).\textit{charAt}(\textit{intg } n), \langle h,l\rangle\rangle \xrightarrow{-} \langle \textit{char } c, \langle h,l\rangle\rangle} \tag{74}$$

$$\frac{h(a) = (ch,m),\ \neg(0 \leq n < m)}{P \vdash \langle (\textit{addr } a).\textit{charAt}(\textit{intg } n), \langle h,l\rangle\rangle \xrightarrow{-} \langle \texttt{THROW IndexOutOfBoundsException}, \langle h,l\rangle\rangle} \tag{75}$$

$$\frac{h(a) = (ch,m),\ \textit{new-Addr } h = a'}{P \vdash \langle (\textit{addr } a).\textit{getBytes}(), \langle h,l\rangle\rangle \xrightarrow{-} \langle \textit{addr } a', \langle h(a \mapsto \textit{encodeToByte}(ch,m)), l\rangle\rangle} \tag{76}$$

Fig. 13. (Exceptional) expression reduction rules for the `String` data type, where: Function $\textit{initString}(str)$ returns a pair of an array ch containing the charachers of the string literal str and its length m. Function $\textit{concat}(ch_1, ch_2)$ creates a new array of characters where the latter array ch_2 is concatenated to the former one ch_1. Function $\textit{encodeToByte}(ch,m)$ converts the string (ch,m) in a new array of bytes.