A Hybrid Approach for Proving Noninterference of Java Programs

Ralf Küsters*, Tomasz Truderung*, Bernhard Beckert†, Daniel Bruns†, Michael Kirsten† and Martin Mohr†
*University of Trier, Germany
Email: {kuesters,truderung}@uni-trier.de
†Karlsruhe Institute of Technology, Germany
Email: {bernhard.beckert,daniel.bruns,michael.kirsten,martin.mohr}@kit.edu

***Abstract*—Several tools and approaches for proving noninterference properties for Java and other languages exist. Some of them have a high degree of automation or are even fully automatic, but overapproximate the actual information flow, and hence, may produce false positives. Other tools, such as those based on theorem proving, are precise, but may need interaction, and hence, analysis is time-consuming.**

In this paper, we propose a *hybrid approach* that aims at obtaining the best of both approaches: We want to use fully automatic analysis as much as possible and only at places in a program where, due to overapproximation, the automatic approaches fail, we resort to more precise, but interactive analysis, where the latter involves the verification only of specific functional properties in certain parts of the program, rather than checking more intricate noninterference properties for the whole program.

To illustrate the hybrid approach, in a case study we use this approach—along with the fully automatic tool Joana for checking noninterference properties for Java programs and the theorem prover KeY for the verification of Java programs—as well as the CVJ framework proposed by Küsters, Truderung, and Graf to establish cryptographic privacy properties for a non-trivial Java program, namely an e-voting system. The CVJ framework allows one to establish cryptographic indistinguishability properties for Java programs by checking (standard) noninterference properties for such programs.

***Keywords*-language-based security; noninterference; program analysis; code-level cryptographic analysis**

I. Introduction

The problem of checking noninterference properties of programs has a long tradition in the field of computer security and, in particular, in language-based security [1]. A program is called noninterferent (w.r.t. confidentiality) if no information from high variables, which contain confidential information, flows to low variables, which can be observed by the attacker or an unauthorized user. Several tools and approaches exist in the literature for checking noninterference. Some approaches, such as type checking [2], [3], [4], [5], [6], [7], abstract interpretations [8], and program dependency graphs (PDGs) [9], with tools including JIF [10], TAJ [11], Joana [9], as well as tools described in [8] and [12], have a high degree of automation, but they overapproximate the actual information flow, and hence, may produce false positives. Other approaches—such as those based on theorem proving—allow for precise analysis, but need human interaction, and hence, the analysis is often time-consuming (see, e.g., [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24]).

Fully automated tools are often preferable over interactive approaches since with such tools program analysis is typically less time-consuming and might require less expertise. However, if automated tools fail due to false positives and the analysis cannot further be refined by these tools, because, for example, the tools do not allow this or run into scalability problems, the only option for proving noninterference so far is to drop the automated tools altogether and instead turn to fine-grained but interactive, and hence, more time-consuming approaches, such as theorem proving. This "all or nothing" approach is unsatisfying and problematic in practice.

Contributions. In this paper, we therefore propose a tool-independent hybrid approach, which allows one to use (fully) automated verification tools for checking noninterference properties as much as possible and only resort to more fine-grained, but possibly interactive verification tools (typically theorem provers) at places in a program where necessary. The latter verification requires checking specific functional properties in (parts of) the program only, rather than checking the more involved noninterference properties.

Our hybrid approach is stated and proven for the language Jinja+ [25], [26], a rich fragment of Java. The simple but powerful idea underlying this approach is as follows. If the verification of noninterference of a program using an automated tool fails due to (what we think are) false positives (i.e., the automated tool falsely claims some illegal flow of information), then, following rules of our approach, additional code is added to the program in order to make it more explicit and more clear for the automated tool that there is no illegal information flow, and by this, avoid false positives. If the automated tool now establishes that the extended program enjoys the desired noninterference property, it remains to show that the extended program is what we call a *conservative* extension of the original program. Intuitively, this means that the additional code did not change the behavior of the original program in an essential way. Proving that an extension is conservative requires to prove *functional* properties of (parts of) the program and will

typically be carried out by an (interactive) theorem prover. The key property that we show for the hybrid approach to work is that if the extended program is noninterferent and is a conservative extension of the original program, then the original program is noninterferent as well. We note that in this work we are concerned with termination-insensitive noninterference since this suffices to prove cryptographic properties for Java programs (see below). However, the basic idea should also be applicable to termination-sensitive noninterference.

While our hybrid approach should be widely applicable—it is not tailored to specific tools or specific applications, and the basic idea is quite independent of a specific programming language—, our main motivation stems from the problem of checking, on the code level, cryptographic properties of programs (that use cryptography), where here we consider Java programs. This has become an active field of research in the last few years (see, e.g., [25], [27], [28], [29], [30], [31] for some of the recent works). More specifically, we use the CVJ framework (cryptographic verification of Java programs) proposed by Küsters, Truderung, and Graf [25] for this purpose. This framework enables tools that can check (standard) noninterference properties for Java programs, but a priori cannot deal with cryptography, to establish cryptographic indistinguishability properties for Java programs. The CVJ framework combines techniques from program analysis and universal composability [32], [33], [34]. Given a Java program (that uses cryptography), the idea is to first check noninterference for this program where cryptographic operations (such as encryption) are performed within so-called ideal functionalities, in the sense of universal composability. The cryptographic framework then guarantees that the actual Java program, where the ideal functionalities are replaced by the actual cryptographic operations, enjoys cryptographic indistinguishability properties.

In order to illustrate our hybrid approach, in a case study we use this approach and the CVJ framework to establish cryptographic privacy properties for a non-trivial Java program, namely an e-voting system. In addition to human actors (voters, auditors, clerks), the system involves a voting machine, which gathers all votes and calculates the election result, and a bulletin board on which the result and other information of the election is published. Now, in order to establish cryptographic privacy properties for this Java system it suffices, according to the CVJ framework, to verify that this system (when run using ideal functionalities for public-key encryption and digital signatures) is noninterferent. To establish noninterference, we use the fully automated tool Joana [9]. However, this tool produces a false positive. (Since establishing noninterference for our case study requires quite intricate reasoning, it seems that all fully automated tools would fail.) We therefore extend the system to avoid the false positive following our hybrid approach, have Joana prove that this extension indeed is noninterferent, and finally prove that the extension is conservative, using the software verification system KeY [35], which is based on theorem proving for dynamic logics. By our hybrid approach, this implies that the system (running with ideal functionalities) is noninterferent. The CVJ framework then immediately yields cryptographic privacy of the Java system when the ideal functionalities are replaced by the actual cryptographic implementations.

Structure of the paper. We first briefly recall some preliminary definitions in Section II, before we present the hybrid approach in Section III. For our case study, we also briefly recall the CVJ framework in Section IV and, in Section V, provide some more background on the tools Joana and KeY that we use. The case study, including the description of the e-voting system and the verification process, is then presented in Section VI. We conclude in Section VIII. Some more details and proofs are provided in the appendix. The source code of the e-voting system, all proofs scripts, and machine generated proofs are available online [36].

II. Preliminaries

We briefly recall the notion of noninterference and also present the Java-like language Jinja+, for which the hybrid approach as well as the CVJ framework [25] are stated and proven.

Jinja+. The Java-like language *Jinja+* [25] is based on *Jinja* [26] and extends this language with some additional features that are relevant in practical applications.

Jinja+ covers a rich subset of Java, including classes, inheritance, (static and non-static) fields and methods, the primitive types `int`, `boolean`, and `byte` (with the usual operators for these types), arrays, exceptions, and field/method access modifiers, such as `public`, `private`, and `protected`. It also includes a primitive `randomBit()` that returns a random bit each time it is called.

A (Jinja+) *program/system* is a set of class declarations. A class declaration consists of the name of the class, the name of its direct superclass, a list of field declarations, and a list of method declarations. A program/system is *closed* if it uses only classes/methods/fields declared in the program itself. A program is *deterministic* if it does not use the `randomBit()` primitive. Note that in fact if `randomBit()` is not used in a program, then up to reference renaming the run of a program is deterministic. (Names of references do not have any effect on the behavior of the program.) For the reader's convenience, we recall the syntax and semantics of Jinja+ programs in Appendix B.

All Java programs considered in this paper, including the systems considered in our case study as well as the ideal functionalities and their realizations, fall into the Jinja+ fragment.

Noninterference. The (standard) noninterference notion for confidentiality [37] requires the absence of information flow

from high to low variables within a program. Here, we define noninterference for closed deterministic (Jinja+) programs where some static variables of primitive types are labeled as high and some other static variables of primitive types are labeled as low. We denote such programs by $P[\vec{x}]$ where $\vec{x}$ is a vector of the variables that are labeled as high in P. For simplicity of notation, the variables in $P[\vec{x}]$ that are labeled as low are omitted in the notation. By $P[\vec{a}]$ we denote the program P where the high variables $\vec{x}$ are initialized with the values $\vec{a}$ and the low variables are initialized as specified in P.

Now, noninterference for a closed deterministic program is defined as follows: Let $P[\vec{x}]$ be a program as above. Then, $P[\vec{x}]$ *is noninterferent/has the noninterference property* if the following holds true: for all $\vec{a}_1$ and $\vec{a}_2$ (of appropriate type), if $P[\vec{a}_1]$ and $P[\vec{a}_2]$ terminate, then at the end of their runs, the values of the low variables are the same. Recall that variables marked low or high are assumed to be static variables of primitive types and note that the above defines *termination-insensitive* noninterference.[1] This is sufficient for applications concerned with cryptographic security properties (see Theorem 3). More general definitions of non-interference can for example be found in [38].

III. A Hybrid Approach for Proving Noninterference

In this section, we present our hybrid approach. As already mentioned in the introduction, this approach provides a method to leverage the precision of certain tools, such as theorem provers (e.g., KeY), in order to allow less precise, but automated tools, such as Joana, to prove noninterference of programs which otherwise they would not be able to deal with.

A. Outline of the Approach

The hybrid approach can be used when an automated tool is employed to prove noninterference of a given program P but the tool reports (as we believe) a false positive, i.e., illegal information flow in a point of a program where (again, as we believe) there is no such illegal flow. Clearly, at this point, one option could be to try to use a more precise, but interactive tool, and drop the automated tool altogether. This would, however, typically involve a substantial, potentially unacceptable amount of work. Our hybrid approach opens up another possibility which combines the advantages of i) automated, but imprecise, and ii) precise, but interactive tools. When applying our approach, one still uses the automated tool as much as possible and resorts to other, more interactive tools only at places in the program where necessary, namely where the automated tool failed.

[1] Also note that in this definition we do not consider low input. However, such low input will be captured in the stronger notion of I-noninterference, where we consider and universally quantify over arbitrary environments (adversaries). These environments provide arbitrary low input.

Our hybrid approach works as follows. Given a program P as above, for which an illegal information flow is reported by the automated tool, we first provide an extension P' of P. We do this following the rules (explained below) of our approach in such a way that (a) it is made (more) explicit for the automated tool that there is no illegal information flow in P' and (b) P' extends P in what we call a *conservative* way. Now, one uses the automated tool to verify that P' is noninterferent. If the automated tool should still fail to prove noninterference for P', because of another (as we believe) false positive, one can further extend P' in a conservative way, and so on. Once, noninterference of P' is established, it remains to verify that P' is in fact a conservative extension of P'. As we will see below, being a conservative extension is a specific functional property. To prove this property, we typically need support of a more precise, but possibly interactive tool, e.g., a theorem prover. However, this should typically involve analyzing only a smaller fragment of the overall program. Being a functional property, this approach is more practical than to prove noninterference properties (of the complete program). If now noninterference and conservatism of P' is established, our hybrid approach (see Theorem 1) implies noninterference of the original program P.

In a nutshell, to construct a (conservative) extension of a program P, one adds an additional component M to the program P. This component is constructed in such a way that its state is isolated from the state of P. The component M is then used to collect some low data and explicitly "kill" potential illegal information flow paths, as explained below.

B. Formalizing the Hybrid Approach

In the following definition, we consider a deterministic and closed program $P[\vec{x}]$, with the variables $\vec{x}$ labeled as high, as in the definition of noninterference. In what follows, we first define the notion of an extension of such a program and then the notion of a conservative extension. We then state the main theorem, which says that if the conservative extension of a program enjoys noninterference, then so does the original program.

Definition 1 (Extension)**.** Let $P = P[\vec{x}]$ be a deterministic and closed (Jinja+) program. An *extension of* P is a program $P' = P'[\vec{x}]$ obtained from P in the following way. First, a new component M is added to P consisting of some number of classes with the following properties:

(i) the methods and fields of the classes in M are static,
(ii) the arguments and the results of the methods of M are of primitive types,
(iii) the methods of M do not refer to classes defined in P (in particular, no methods and fields of P are used in M),
(iv) all potential exceptions are caught inside M,
(v) all methods of M always terminate.

Second, P is extended by adding statements of the following form in arbitrary places within methods of P:

(a) (*output to M*)

$$C.f(e_1,\ldots,e_n) \tag{1}$$

where C is a class in M with a (static) method f and $e_1,\ldots,e_n$ are expressions without side effects.

(b) (*input from M*)

$$r = C.f(e_1,\ldots,e_n), \tag{2}$$

where C is a class in M, $C.f$ is a (static) method with some (primitive) return type τ, $e_1,\ldots,e_n$ are expressions as above, and r is an expression that evaluates without side effects to a reference of type τ. (Such an expression can, for example, be a variable or an expression of the form $o.x$, where o is an object with field x.) □

The property of being an extension of a program typically is easy to verify. Conditions (i) to (iii) can easily be checked syntactically. As for Condition (iv), typically M should be a quite simple class, and hence, it should be easy to see that methods cannot throw exceptions. If there are potentially problematic operations, one can explicitly catch them inside M. Whether this is done can, again, be checked syntactically. Similarly, for Condition (v) there are simple syntactical criteria such as a lack of loops and recursion. These criteria seem to suffice for the typical ways in which conservative extensions are constructed, as we expect them to be simple.

Note that an extension is defined in such a way that the state of the added component M is separated from the state of P (these components do not share references; see Appendix A for a formal definition of state separation). Moreover, additional statements of the form (1) do not change the state of P. Only statements of the form (2) could potentially change the state of P. We call an extension conservative if this, however, is not the case:

Definition 2 (Conservative extension)**.** An extension $P'[\vec{x}]$ of $P[\vec{x}]$ is called a *conservative extension* of $P[\vec{x}]$, if for all initial values $\vec{a}$ of high variables $\vec{x}$ the following is true in the run of $P'[\vec{a}]$: Whenever a statement of the form (2) is executed, it does not change the value of r. That is, the value of r right *before* the execution of the assignment coincides with the value returned by the method call $C.f(e_1,\ldots,e_n)$. As such, statement (2) is redundant. □

The intuition behind the above definitions is the following. In a conservative extension P' of P, not only the state of M is disjoint from the state of the original program P, but additionally all the added statements preserve the state of P (and hence are redundant). Altogether, P and its conservative extension P' are equivalent in that they produce the same runs up to calls to M and up to the state of M.

Being a conservative extension obviously is a functional property. Proving this property requires a (possibly non-trivial) proof. For such a proof one would typically use an (interactive) theorem prover. However, for the hybrid approach it clearly does not matter by what means conservatism is established. If this property can be established by some automated tool (probably a different tool than the one used for proving noninterference of P'), then one can of course use such a tool.

As illustrated by our example below and our case study in Section VI, proving conservatism will typically be something that one implicitly would have to prove even if one tried to prove noninterference directly for the original program P: In the conservative extension P' certain information flows are canceled out explicitly. This will typically be one of the "puzzle pieces" for the noninterference of P anyway. So, the hybrid approach typically will not add proof obligations.

Now, following the intuition that the runs of a program P and a conservative extension of P produce the same runs up to calls to M and the state of M, we prove the following theorem (see Appendix A for the full proof). The theorem says that if a conservative extension of a program with high and low variables is noninterferent, then so is the program itself. Note that part of the proof of the theorem is that the conditions for M in Definition 2 in fact guarantee that M does not change the state of a program.

Theorem 1. *Let $P[\vec{x}]$ be a program with the variables $\vec{x}$ labeled as high and variables $\vec{y}$ labeled as low. Let $P'[\vec{x}]$ be a conservative extension of $P[\vec{x}]$ such that in $P'[\vec{x}]$ again the variables in $\vec{x}$ are labeled as high and those in $\vec{y}$ are labeled as low. Then, if $P'[\vec{x}]$ is noninterferent, then so is $P[\vec{x}]$.*

Proof sketch: Let $P'[\vec{x}]$ be a conservative extension of $P[\vec{x}]$. Recall that the high variables $\vec{x}$, as well as the low variables $\vec{y}$, are assumed to be static and of primitive types. Notice also that these variables do not occur in M (the additional class of P').

Now, suppose that the program $P'[\vec{x}]$ is noninterferent, whereas the original program $P[\vec{x}]$ is not. This means that there exist values $\vec{a}_1$ and $\vec{a}_2$ such that $P[\vec{a}_1]$ and $P[\vec{a}_2]$ terminate and for the final values $\vec{b}_1$ and $\vec{b}_2$, respectively, of the low variables $\vec{y}$ we have that $\vec{b}_1 \neq \vec{b}_2$.

Clearly, the variables $\vec{y}$ are part of the state of the original program P. Therefore, one can show that, due to the separation of states of the original program P and the state of M and because the additional statements added to P do not change the state of P, the programs $P'[\vec{a}_1]$ and $P'[\vec{a}_2]$ (1) terminate as well and, moreover, (2) terminate with the same final values of the low variables $\vec{y}$, namely with $\vec{b}_1$ and $\vec{b}_2$, respectively. Since $\vec{b}_1 \neq \vec{b}_2$, this, however, means—contrary to what we have assumed—that the program $P'[\vec{x}]$ is not noninterferent. Hence we have reached a contradiction and proven that $P[\vec{x}]$ is noninterferent. We refer the reader to Appendix A for the full proof. ■

We refer the reader to our technical report for the full

proof.
The definitions and results stated above do not say how one can come up with a conservative extension of a program. This process is not automatic and requires some understanding of the analyzed program and the automated tool that is used.[2] We now explain this process.

C. Constructing a Conservative Extension

As sketched above, the approach to come up with a conservative extension of a program is as follows: For the sake of the discussion, say that some potentially high information flows into some variable v and from this point this information flows to some low variables. If the value assigned to v does not actually depend on high variables, the conservative extension should make this more explicit. For this purpose, the component M should be provided with low data that is sufficient to compute the value for v. This is done by adding statements of the form (1) in Definition 1 (output to M) in such a way that it is apparent for the automated tool that the data given to M is low. Then, at the problematic point in the program (typically after the point in the original program where v is assigned a value) an assignment to v of the form (2) in Definition 1 (input from M) is added to the program to make it more explicit for the automated tool that v depends on low data only. In other words, the assignment explicitly "kills" potential dependencies from high data.

We illustrate this general idea by a toy example, with a more interesting and complex case study presented in Section VI. More specifically, by the example presented below we illustrate the process of running into a false positive with an automated tool, extending the code, proving that this extension has the desired noninterference property (using some automated tool), and proving that the extension is conservative (by some interactive theorem prover). The example also demonstrates that the hybrid approach can be very beneficial, compared to giving up on automated analysis altogether and switching to an interactive theorem prover for checking noninterference for (the whole) program.

Example. We consider the following Java program, where `secret` is declared to be a high variable and `result` is low.

```
1  class Example {
2      static public int result;
3      static private int a;
4      public static void main(int secret) {
5          a = 42;
6          bar(secret);
7          int b = foo(secret);
8          result = b;
9      }
10     static int foo(int secret) {
11         int b = a;
12         // M.set(a);
13         if (secret==0) b+=secret;
14         // b = M.get();
15         return b;
16     }
17     static void bar(int secret) {
18         ...
19     }
20 }
21
22 /*
23 class M {
24         static int x;
25         public static void set(int n) { x=n; }
26         public static int  get()      { return x; }
27 }
28 */
```

We applied the fully automated tool Joana for checking noninterference for Java programs to this example (see Section V-A for more information about Joana), where `bar` was some piece of code on which Joana did not report an illegal information flow. Joana reported in method `foo`, line 13 a potential information flow from the variable `secret` to the variable `b`, the value of which is then returned and assigned to the low variable `result`. This flow is, however, not real, as can easily be seen: the value of `b` is in fact *not* altered in line 13 and it only depends on the low variable `a`. We can easily make this nondependency explicit by adding (uncommenting) the class `M` and uncommenting lines 12 and 14. Now, Joana establishes noninterference for this extended program without problems.[3] Note that, in particular, Joana tells us that the method `bar` does not cause `a`, and hence, `result` to depend on high information.

It remains to prove that the extension is conservative, i.e., that before executing line 14 the variable `b` carries the value returned by `M.get()` (= `a`). This is merely a functional property which can easily be verified using the theorem prover KeY (see Section V-B for more background on KeY). Note that in order to prove this property, KeY has to analyze only the code of the method `foo` and the class `M`. In particular, it does not have to touch the potentially large method `bar` or other methods or classes that might be contained in the program. So the analysis one has to perform with the theorem prover, KeY in this case, is relatively simple in that for checking the functional property only part of the program has to be touched (rather than the whole program) and checking the functional property would be part of a full-fledged noninterference proof anyway.

Now, by the results of Joana and KeY, Theorem 1 implies that our original example program (with the class `M` and

[2]Note that proving noninterference properties for the whole program, rather than functional properties for parts of the program, with an interactive tool would typically require much more understanding of the program and human interaction. Also, and importantly, as explained, typically proving conservatism would implicitly be part of the proof of noninterference of the original program anyway.

[3]We note that for this the fact that Joana is flow sensitive is important. Without this property, Joana would not see that line 14 overwrites the value of b in line 13.

lines 12 and 14 commented out) enjoys the noninterference property, a fact that with Joana alone could not have been established.

We note that even in complex applications, M will often be very simple. Therefore, to ease the notation, one can often safely inline the code of methods of M. In the above example, one could replace lines 12 and 14 by `M.x = a` and `b = M.x`, respectively. In some cases, one can even move static fields of M to the original program and get rid of M completely. In our example, we could obtain an equivalent variant by declaring x to be a static field of class `Example` and replace lines 12 and 14 by `x = a` and `b = x`, respectively.

IV. Framework for Cryptographic Verification of Java Programs

As already mentioned in the introduction, while the hybrid approach should be widely applicable (it is not tailored to specific tools or applications), our main motivation for devising methods for proving noninterference properties comes from the problem of establishing cryptographic guarantees, in particular, cryptographic indistinguishability, for Java programs.

In [25], a framework, which is referred to as the *CVJ framework*, for cryptographic verification of Java programs, has been proposed which enables tools that can check (standard) noninterference properties for Java programs, but a priori cannot deal with cryptography (probabilities, polynomially bounded adversaries), to establish cryptographic indistinguishability properties of Java programs. For this purpose, the framework combines techniques from program analysis and universal composability. Given a Java program that uses cryptographic operations, the framework shows that in order to verify that the program enjoys a cryptographic indistinguishability property it suffices to prove, using the tools, that the program enjoys a (standard) noninterference property when the cryptographic operations are replaced by so-called ideal functionalities. The reason for using ideal functionalities is that they often do not involve probabilistic operations and are secure even for unbounded adversaries, which are the kind of adversaries considered for standard noninterference properties. The framework has been stated and proven in [25] for the language Jinja+ (see Section II).

In this section, we briefly recall the CVJ framework. The definitions and theorems stated here are somewhat simplified and informal, but should suffice to follow the rest of the paper. We refer the reader to [25] for full details.

Indistinguishability. An *interface* I is defined like a (Jinja+) system but where (i) all private fields and private methods are dropped and (ii) method bodies as well as static field initializers are dropped. A system S *implements* an interface I, written $S : I$, if I is a subinterface of the public interface of S. 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 $S : I_1$.

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* (or *adversary*) 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 closed program. The value written by E to `result` at the end of a 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, referred to as *probabilistic polynomially bounded systems*. We omit the details of the runtime notions used in the CVJ framework, 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 (closed, possibly probabilistic) programs. We say that P_1 and P_2 are *computationally equivalent*, written $P_1 \equiv_{\mathsf{comp}} P_2$, if $|\mathsf{Prob}\{P_1(\eta) \rightsquigarrow \texttt{true}\} - \mathsf{Prob}\{P_2(\eta) \rightsquigarrow \texttt{true}\}|$ is a negligible function in the security parameter η.[4]

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. 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 $S \cdot F$. So, intuitively, the

[4] 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 $\eta > \eta_0$.

simulator tries to make $S \cdot F$ look like R for the environment.

To provide some intuition, consider, for example, public-key encryption. When instructed to encrypt a message m, the ideal functionality F would instead encrypt the message $0^{|m|}$ and it would store the pair (m,c), where c is the ciphertext resulting from encrypting $0^{|m|}$. So, by definition of F, the ciphertext c does not contain any information about the plaintext m, except for its length. If later F is asked to decrypt c, it would look up the corresponding plaintext in its table and return this plaintext, m in this case. Conversely, in R the actual message would be encrypted and for decryption one would simply apply the decryption algorithm to the given ciphertext. One can show that if the cryptographic algorithms used in R are IND-CCA2-secure (a standard security assumption for public-key encryption schemes), then R in fact realizes F. Such a result has been proven in [30] for Java implementations of R and F, both for public-key encryption and digital signatures (with public-key infrastructures). We use these results in our case study in Section VI.

To define the notion of realization more formally, let F and R be probabilistic polynomially bounded systems which implement the same interface I_{out} and use the same interface, except that in addition F may use some interface 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 [25], $\leq$ is reflexive and transitive. It also enjoys composability, i.e., one can analyze a system using ideal functionalities (such as F) and later replace these functionalities by their realizations (such as R); see also Theorem 3.

***I*-noninterference.** The standard notion of (termination-insensitive) noninterference (see Section II) deals with closed programs/systems. For the CVJ framework, this notion is generalized to open systems as follows: Let I be an interface and let $S[\vec{x}]$ be a (not necessarily closed) deterministic system with high variables $\vec{x}$ such that $S : I$. Then, $S[\vec{x}]$ is *I-noninterferent* if for every deterministic I-environment E for $S[\vec{x}]$ and every security parameter η, noninterference holds for the system $E \cdot S[\vec{x}](\eta)$, where the variable `result` declared in E is considered to be the only low variable in the system $E \cdot S[\vec{x}](\eta)$. Note that here neither E nor S are required to be polynomially bounded.

Tools for checking noninterference 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 [25] a technique was developed which reduces the problem of checking I-noninterference to checking noninterference for a single (almost) closed system. More specifically, it was shown that to prove I-noninterference for a system $S[\vec{x}]$ with $I_E \vdash S : I$ it suffices to consider a single environment $\tilde{E}^{I,I_E}_{\vec{u}}$ only (simply denoted by $\tilde{E}_{\vec{u}}$), which is parameterized by a sequence $\vec{u}$ of values.[5] 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 the interfaces between $S[\vec{x}]$ and E. For example, $S[\vec{x}]$ and E should interact only through primitive types, arrays, exceptions, and simple objects.

Theorem 2 (simplified, [25])**.** *I-noninterference, for $I = \emptyset$, holds true for $S[\vec{x}]$ if and only if for all sequences $\vec{u}$ noninterference holds true for $\tilde{E}_{\vec{u}} \cdot S[\vec{x}]$.*

Analysis tools often ignore or can ignore specific values encoded in a program, such as an input sequence $\vec{u}$. So, if such an analysis establishes noninterference for $E_{\vec{u}} \cdot S[\vec{x}]$, with some vector $\vec{u}$, the theorem implies I-noninterference for $S[\vec{x}]$.

From *I*-Noninterference to Computational Indistinguishability. In [25], it was shown that the problem of verifying cryptographic privacy of the secret (high) input given to a system $S[\vec{x}]$ can be reduced to the simpler problem of verifying I-noninterference for $S[\vec{x}]$ where the cryptographic operations performed by $S[\vec{x}]$ are replaced by ideal functionalities.[6] This enables tools that can check (standard) noninterference properties but cannot deal with cryptography (probabilities, polynomially bounded adversaries) to establish strong cryptographic privacy properties. Note that such tools assume unbounded adversaries, which can easily break basically all cryptographic operations, such as encryption. The result in [25] avoids this problem by replacing the cryptographic operations by ideal functionalities, which are secure even w.r.t. unbounded adversaries. More specifically, the following theorem immediately follows from results proven in [25].

Theorem 3 (simplified, [25])**.** *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 is that the cryptographic operations that P needs to perform are carried out using the system R (a cryptographic library). The theorem says that to prove privacy of the secret inputs ($\forall\ \vec{a}_1, \vec{a}_2$: $P[\vec{a}_1] \cdot R \approx^J_{\mathsf{comp}} P[\vec{a}_2] \cdot R$) it suffices to prove I-noninterference for $P[\vec{x}] \cdot F$, i.e., the system where R is replaced by the ideal counterpart F. To prove I-noninterference for $P[\vec{x}] \cdot F$, one can in turn use Theorem 2 to reduce the problem further to checking standard noninterference, which in turn can be done using the hybrid approach if an automated tool alone would fail.

[5] The sequence $\vec{u}$ can be seen as low input to $\tilde{E}$.

[6] The result in [25] is actually more general and considers general cryptographic indistinguishability properties, but here we need privacy properties only.

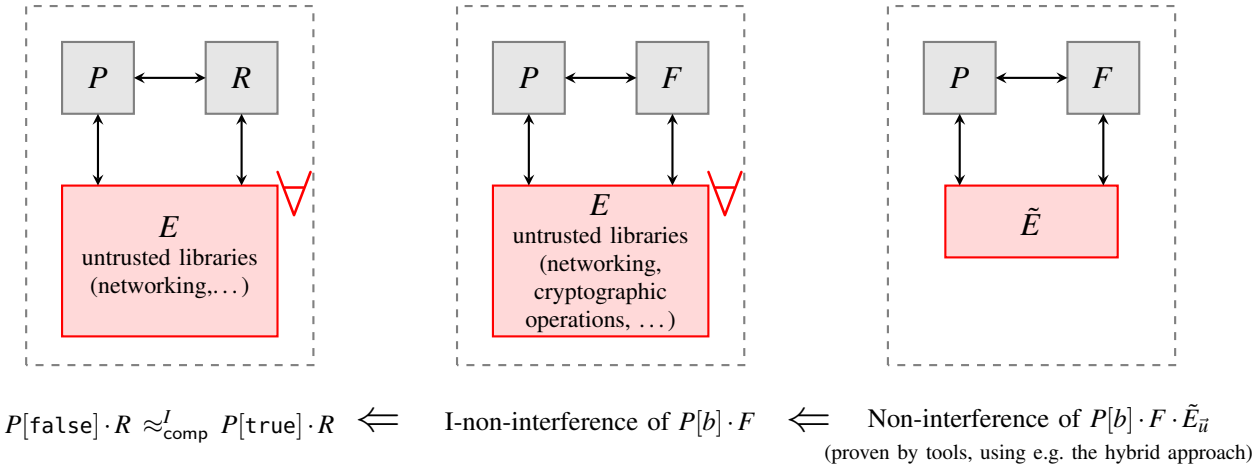

Figure 1. By Theorems 2 and 3, tools for proving noninterference can be used to prove cryptographic privacy of a program P that uses some cryptographic operations (real cryptographic functionalities) R and is connected to untrusted libraries subsumed by the environment E (see the left-most system above). For simplicity of presentation, in this figure we assume P to have only one high boolean variable. By these theorems, to prove cryptographic privacy of this system (that is $P[\texttt{false}]\cdot R \approx^{I}_{\text{comp}} P[\texttt{true}]\cdot R$), it is enough to show (classical) noninterference of the system $P[b]\cdot F\cdot \tilde{E}$, where F are the ideal functionalities corresponding to R and $\tilde{E}_{\vec{u}}$ is the specific environment given in Theorem 2. Note that F often uses cryptographic libraries. However, the cryptographic guarantees are established by F itself (e.g., F might call an encryption function, but with $0^{|m|}$ instead of the actual message m, and hence, by the definition of F and independently of the encryption function, the ciphertext will not reveal information about m, except for the length of m). Therefore, the cryptographic libraries that F uses can be untrusted, and hence, provided by the environment. Conversely, the cryptographic library implemented in R is supposed to realize F. So, this library cannot be subsumed by the environment.

The overall approach is illustrated in Figure 1.

Jumping ahead, in our case study (see Section VI), the Java system $P[\vec{x}]\cdot R$ (and $P[\vec{x}]\cdot F$) describes an indistinguishability or privacy game for an e-voting system, where the adversary has to guess a bit $b \in \{\texttt{true}, \texttt{false}\}$. Hence, the high input $\vec{x}$ is simply b, and we want to prove that $P[\texttt{false}]\cdot R \approx^{I}_{\text{comp}} P[\texttt{true}]\cdot R$. This means that the environment/adversary cannot distinguish whether the secret bit is `false` or `true`.

V. Tools

While our hybrid approach does not depend on any specific tools, for the analysis of concrete systems one has to choose tools to perform the analysis (based on the hybrid approach). In our case study and in the toy example from Section III we use the tools Joana and KeY. In this section, we provide some background about these tools.

A. Joana

Joana[7] [9], [39] is a tool for the fully automatic analysis of noninterference properties of Java programs. A user needs to only specify the high and low variables of a program. Joana is based on the program analysis framework WALA.[8] It computes a conservative approximation of the information flow inside the program in form of a *program dependence graph* (PDG). Then, the PDG is checked for illegal information flows using advanced dataflow analysis based on *slicing*. If no illegal flow is found in the PDG, the program is guaranteed to be noninterferent. The correctness of this implication has been verified with a machine-checked proof [40] which includes formal specifications of PDGs and the slicing algorithm [41], [42].

The fully automatic analysis performed by Joana comes at the cost of potential false alarms due to overapproximation. Joana leverages sophisticated flow-, context-, field- and object-sensitive analysis techniques that help to reduce such false alarms, but it does not consider actual values of variables. For example, whenever a high variable is used in an expression, the value of the expression is considered to contain high information—even if the value of the high variable does not actually influence the result (this phenomenon is also illustrated by the example in Section III and our case study in Section VI).

B. The KeY Verification System

KeY[9] [43], [35], [44] is an integrated program verification system, which targets sequential Java. At its core lies an interactive theorem prover for first-order dynamic logic (JavaDL) [45]. Program specifications can be given in the Java Modeling Language (JML) [46], [47]. KeY provides both a stand-alone graphical user interface, intended for interactive proofs, and an integration into the Eclipse platform,

[7]The sourcecode of Joana and additional information is available at `http://joana.ipd.kit.edu/`.

[8]`http://wala.sf.net/`

[9]KeY is free software and can be downloaded (in source or binary) from http://key-project.org/, the current stable version is 2.4.

intended for push-button proofs hiding the underlying prover architecture [48].

In dynamic logic [49], [50], programs π give rise to modal operators $[\pi]$ and $\langle\pi\rangle$. For instance, the formula $\phi \rightarrow \langle\pi\rangle\,\psi$ intuitively means "if started in a state in which formula ϕ holds, program π terminates, and in the final state, formula ψ holds." This means that the right-hand side of this implication is equivalent to the weakest precondition of π w.r.t. ψ. Replacing $\langle\cdot\rangle$ by $[\cdot]$ yields the weakest liberal precondition. Dynamic logic can be seen as a super-set of Hoare logic [51]. In contrast to Hoare triples, however, programs are an integral part of formulae. This allows one to write down more elaborate formulae, e.g., formulae with multiple programs or existential quantification ranging over program states.

The sequent calculus for JavaDL that is built into KeY precisely reflects the semantics of sequential Java, i.e., it does not use approximations. Thus, analysis techniques built on KeY are precise. They do not report any false positives. Proofs can be automated to a certain degree, while the user can interact with the prover at any time. KeY can generate counter examples and unit tests from failed proof attempts.

VI. The Case Study

To illustrate that our hybrid approach can successfully be used to verify non-trivial programs, we applied our approach to prove that a Java implementation of an electronic voting system provides cryptographic vote privacy.[10] This system, even if relatively simple for an e-voting protocol, is challenging from the point of view of code-level cryptographic verification in that it appears impossible to completely verify this system using just fully automated tools (i.e., tools that hardly need any human input or interaction) for checking noninterference. Also, we do not see a feasible way to verify noninterference for this Java program directly with a theorem proving approach given the currently available tools. At the very least, the hybrid approach, tremendously reduces the necessary effort.

In what follows, we first provide a brief description of the analyzed program. We then state the cryptographic privacy property that we verify and describe the verification process, which is based on our hybrid approach, using the tools Joana and KeY, and the CVJ framework. The code of the analyzed system, as well as the proof scripts, are available under [36].

A. Description of the Analyzed System

We consider an electronic voting system where voters cast their ballots in a polling station. The system involves a voting machine and a bulletin board as well as human actors, namely voters, auditors, and clerks. The voting machine and the bulletin board are implemented in Java, including the real/ideal cryptographic operations that they use. Since, in the end, we analyze Java code, the human actors will be modeled as Java programs as well. More specifically, they will be modeled in the privacy game which is formulated as Java code and expresses the privacy property we are interested in. This game is described in Section VI-B.

In what follows, we describe the expected behavior of all components (the bulletin board, the voting machine, and the human actors) and their interaction. Before going into more detail, we first provide a brief overview of the actions of the bulletin board and the voting machine, which use public-key encryption and digital signatures.

The bulletin board allows the voting machine to publish messages in a write-only manner (no messages can be altered or deleted after they have been posted). The bulletin board accepts messages only if they are signed by the voting machine. The messages published on the bulletin board are made public to any party. More specifically, parties can ask the bulletin board to deliver its content to them.

The voting machine collects voters' choices and publishes the result and some additional data on the bulletin board. The voting machine communicates with the bulletin board over an untrusted network, which is controlled by the adversary. In its internal state, the voting machine stores the current partial result, i.e., for each possible election choice (candidate) an integer representing the number of votes cast so far for this choice. In addition, the machine maintains an operation counter. That is, an integer which is incremented every time the machine performs one of its critical operations, such as a vote is collected or canceled (see below). Moreover, the voting machine keeps an encrypted log of all critical steps. For this purpose, the voting machine uses the public-key of the auditors so that the log can be decrypted by them only.

Setup phase. The auditors, among whom at least one should be honest, publish a public encryption key and share the corresponding private decryption key in such a way that cooperation of all the auditors is necessary to decrypt a message. Hence, auditors can only decrypt messages collectively. In our modeling, for simplicity we will assume one auditor and this auditor should be honest.

Voting. We assume that clerks, among whom one should be honest, check the identity of each voter and make sure that only authorized voters can vote and that each voter votes only once. Voters indicate their choices directly to the voting machine.

When a voter submits her choice (candidate), the voting machine collects this choice (that is, increases the counter associated with this choice). Additionally, the voting machine performs the following steps, which are meant to allow for later auditing. First, the voting machine increases its operation counter. Then, the machine encrypts the voter's choice, plus some additional data (such as a time stamp and an appropriate tag), under the public key of the auditors. An entry that consists of the current value of the operation

[10] While privacy is not the only property one would require from an e-voting system, it is a central one.

counter and the ciphertext just created is then logged by the machine in its internal log. In addition, the machine signs this entry (using the machine's signing key) and sends it to the bulletin board. Finally, the voting machine returns an operation counter to the voter.

Vote canceling. Auditors (together) can decide to vote just like a voter. In particular, this action is logged, internally and on the bulletin board, as described above. The auditors check that an entry that carries the same operation counter as the one given to the auditors when they voted is indeed added to the bulletin board. By asking the machine to output the internal log (as described below), the auditors can also make sure that the voting machine logged this entry internally.

Right after voting, the auditors are supposed to indicate to the voting machine to cancel their vote. We assume that the canceling operation is available to auditors only (canceling might require some physical key or password). Upon such a canceling request, the voting machine first decrements the counter corresponding to the election choice made by the auditors (i.e., the last choice indicated to the machine). Also, the machine increments the operation counter. Finally, this action is logged internally and on the bulletin board similarly to the action of voting.

Log publishing. At any time, the machine may be triggered by the auditors to publish the complete internal (encrypted) log. This may be useful, when, for instance, some of the messages sent by the voting machine in the voting step got lost because of network problems.

Result publishing. Finally, when the voting phase is over, the voting machine can be triggered by the auditors (or clerks) to publish the election result. The machine signs the result and sends it to the bulletin board.

We note that certain entries or even the complete logs (internal and on the bulletin board) could be decrypted by the auditors to verify actions of the voting machine. Typically, auditors would open entries corresponding to their voting and cancellation actions. However, they could also decide to open a (small) sample of other entries in the logs in order to statistically compare this sample with the published result. Of course this would weaken privacy. While in our analysis, we take vote canceling operation into account, for simplicity of the analysis we do not model the orthogonal issue of opening of entries by auditors.

B. The Privacy Game

We want to verify that the (Java implementation of the) above e-voting system ensures cryptographic privacy of the votes of honest voters. For this purpose, we define a cryptographic privacy game, similar to games in cryptography, except that this game is formulated in Java. It is this game we have to analyze, using the hybrid approach, the CVJ framework, and the tools Joana and KeY.

To define this game, let ρ be a result function which takes a multiset (or a vector) of choices/candidates and returns a result vector $\vec{v}$, i.e., for every choice, $\vec{v}$ contains an entry with the number of occurrences of this choice in the given multiset.

In a nutshell, in the privacy game the environment (the adversary) can provide two vectors $\vec{c}_0$ and $\vec{c}_1$ of choices of (honest and dishonest) voters such that the two vectors yield the same result according to ρ, i.e., $\rho(\vec{c}_0) = \rho(\vec{c}_1)$; otherwise the game is stopped immediately. Now, the voters vote according to $\vec{c}_b$, where b is a secret bit. The environment tries to distinguish whether the voters voted according to $\vec{c}_0$ or to $\vec{c}_1$. In other words, the environment tries to determine b. We denote the Java program describing this game by $\mathsf{EV}_{Real}[b]$, with b being the only secret/high input. Note that for the considered e-voting system there is no difference between an honest and a dishonest voter. Just like for honest voters, for dishonest voters the clerks check eligibility and make sure that these voters vote at most once, then they can only cast one vote which they have to indicate to the machine. Hence, our analysis takes dishonest voters into account.

More specifically, the Java program $\mathsf{EV}_{Real}[b]$ contains a setup class with the method `main`, which works as follows. First, as mentioned, the environment (the adversary) is asked by `main` to provide two vectors $\vec{c}_0$ and $\vec{c}_1$ of n (valid) choices, one for each voter. Then, `main` checks that $\rho(\vec{c}_0) = \rho(\vec{c}_1)$. If this is not the case, `main` aborts, and hence, the privacy game halts. Otherwise, the voting protocol is performed where the voters vote according to $\vec{c}_b$. More specifically, `main` creates one object for the voting machine and one object for the bulletin board. Now, in a loop, the environment decides which actions are executed and it also determines how long the loop is executed. Determining the actions corresponds to oracle queries in cryptographic security games. The actions the environment can trigger include the following:

- triggering the next voter to vote,
- triggering the auditors to cast a vote (determined by the environment) and then cancel it,
- triggering the voting machine to output the content of the internal log (with encrypted entries).
- posting a message determined by the environment on the bulletin board, and
- reading the content of the bulletin board.

If all voters have voted, `main` triggers the voting machine to publish the election result on the bulletin board. As usual (see also Section IV), the environment is supposed to output its decision, i.e., its guess of b, in a distinct private static (low) variable `result`.

In the above game, the environment controls the network. More precisely, the network library, which all components use, is provided by the (malicious) environment (see also the box on the left-hand side of Figure 1). Hence, if the network library is invoked to send a message to a party, this

message is directly given to the environment. Also, if a party wants to read a message from the network, this message is provided by the environment. In particular, all data that the voting machine sends to the bulletin board, since it is sent via the network, is given to the environment.

For public-key encryption and digital signatures, the bulletin board and the voting machine use Java classes proposed in [30], which support a public-key infrastructure. As shown in [30], under standard cryptographic assumptions (IND-CCA2 security of the public-key encryption scheme and UF-CMA security of the digital signatures scheme) these classes realize suitable ideal functionalities. This is used later in our analysis (see below).

C. The Security Property

As mentioned before, the security property we want to verify for the e-voting system is the cryptographic privacy of the votes of honest voters. Formally, this is expressed by the computational indistinguishability property (see Section IV) as follows:

$$\mathsf{EV}_{Real}[\texttt{false}] \approx^{\emptyset}_{\mathsf{comp}} \mathsf{EV}_{Real}[\texttt{true}]. \tag{3}$$

This property says that a polynomially bounded environment (adversary) is not able to distinguish whether the privacy game EV_{Real} for the e-voting system it interacts with uses `true` or `false` as the initial value of the variable b. By the definition of EV_{Real}, this in turn means that even though the environment dictates the two possible ways in which the voters may vote (as long as they yield the same result), the environment is not able to tell which one of them has actually been used (except with negligible probability).

The computational indistinguishability relation in (3) uses the empty interface $I = \emptyset$. This means that the environment cannot directly call methods defined in the e-voting system. However, the environment controls the network and by definition of EV_{Real} (in particular, `main`) it can determine which actions are taken and when.

We also note that EV_{Real} is an open system, which uses some classes not defined within EV_{Real}, in particular for the network library. These classes are provided by the environment,[11] and hence, are untrusted and controlled by the environment. Thus, property (3) implies privacy of honest votes no matter how such untrusted libraries are implemented. In particular, as mentioned before, this also means that the environment controls the network.

As already explained in Section IV and illustrated in Figure 1, in order to prove (3), by Theorem 3 it suffices to show that

$$\mathsf{EV}_{Ideal}[b] \text{ is } I\text{-noninterferent}, \tag{4}$$

where $I = \emptyset$ and EV_{Ideal} denotes the system which coincides with EV_{Real} except that the real cryptographic operations are replaced by their ideal counterparts. More specifically, the real cryptographic operations for public-key encryption and digital signatures are replaced by ideal functionalities for these primitives, as provided in [30]. The realization result in [30] shows that the real cryptographic operations realize the ideal functionalities. Hence, these operations can indeed be replaced by their ideal counterparts.

Since, as can easily be seen, $\mathsf{EV}_{Ideal}[b]$ satisfies the conditions of Theorem 2, we can further reduce checking (4) to checking the following property:

$$\tilde{E}_{\vec{u}} \cdot \mathsf{EV}_{Ideal}[b] \quad \text{is noninterferent for all } \vec{u}, \tag{5}$$

where the family of systems $\tilde{E}_{\vec{u}}$, parameterized by a finite sequence of integers $\vec{u}$, is as described in Section IV. This system can be automatically generated from $\mathsf{EV}_{Ideal}[b]$. Also note that by "noninterference" we mean standard termination-insensitive noninterference (see Section II).

Altogether, it suffices to prove (5) in order to obtain (3). In the following, we will refer to the system $\tilde{E}_{\vec{u}} \cdot \mathsf{EV}_{Ideal}[b]$ by $\mathsf{EV}_{\vec{u}}[b]$.

D. Verification Approach

In principle, the automated tool Joana is able to check properties such as (5) (see, e.g., [25], [30]). However, when applied to check (5) for our particular program, Joana reports an information flow from the high value `b` to the result of the election, and from this result to the low output.[12] The reason for this alert is the overapproximation that Joana employs. The election result actually does not depend on `b`, because `main` in $\mathsf{EV}_{\vec{u}}[b]$ ensures that the vectors $\vec{c}_0$ and $\vec{c}_1$ produce the same election result. If the voting machine computes the result of the election correctly, then this result is independent of whether the voters voted according to $\vec{c}_0$ or to $\vec{c}_1$. Hence, to avoid this false positive, an analysis tool has to establish that the voting machine correctly computes the result of the election and that this result corresponds to $\vec{c}_0$ (and hence, $\vec{c}_1$). This is a non-trivial functional property, which — not surprisingly — is beyond what Joana and, it seems, all other fully automated tools for checking noninterference can achieve.

We therefore use our hybrid approach. It is straightforward to provide an extension $\mathsf{EV}^*_{\vec{u}}[b]$ of $\mathsf{EV}_{\vec{u}}[b]$ which makes it more explicit for Joana that there is no information flow. If the voting machine indeed works as expected, the election outcome corresponds to $\vec{c}_0$ (and hence, $\vec{c}_1$). So in the extension $\mathsf{EV}^*_{\vec{u}}[b]$ we explicitly state that the election outcome actually *is* the result corresponding to $\vec{c}_0$.

More specifically, we obtain $\mathsf{EV}^*_{\vec{u}}[b]$ from $\mathsf{EV}_{\vec{u}}[b]$ as follows: The result of the election as determined by the vector

[11] Recall from Section IV that an I-environment E for a system S provides all classes that S needs but are not defined in S in order for the composed system $E \cdot S$ to be a closed (and hence, runnable) system.

[12] We note that the output produced by Joana allows one to conveniently identify potential flows of information.

$\vec{c}_0$ is stored in an additional static array `correctResult` in $\mathsf{EV}^*_{\vec{u}}[b]$ which contains the number of votes for each candidate as determined by $\vec{c}_0$. (This field is the only field of extension M, as introduced in Definition 2). Moreover, after the point in the code of the voting machine where the number of votes for the i-th candidate is computed by the voting machine and stored in a local variable `x`, we add the assignment `x = correctResult[i]`.

If the system is implemented correctly, then the result computed by the voting machine (the number of votes for each candidate) indeed is the same as the result stored in `correctResult` and, therefore, the additional assignment does not change the state of the program. In fact, we successfully used the KeY tool to verify that in the state just before the additional assignment, the variable `x` already has the value of `correctResult[i]`, a statement we will refer to by (*). This means that $\mathsf{EV}^*_{\vec{u}}[b]$ is a conservative extension of $\mathsf{EV}_{\vec{u}}[b]$. (It is trivial to see that our extension M satisfies all the conditions required by Definition 1.)

We emphasize that proving (*) would be part of every proof of noninterference: if, for example, due to programming errors the voting machine would drop or miscalculate some of the votes, an adversary might be able to distinguish whether the voters voted according to $\vec{c}_0$ or $\vec{c}_1$. Also (*) is a functional property of (part of) the program rather than a noninterference property (of the whole program). In order to prove (*), many things, which would be important for noninterference, do not need to be taken into account. For example, one can prove (*) even if the voting machine revealed votes, provided that the final result is computed correctly.

E. Proving Noninterference with Joana

For the extension $\mathsf{EV}^*_{\vec{u}}[b]$ of $\mathsf{EV}_{\vec{u}}[b]$, Joana easily established the noninterference property:

$$\mathsf{EV}^*_{\vec{u}}[b] \quad \text{is noninterferent for all } \vec{u}. \tag{6}$$

Joana took about 18 seconds on a standard PC (Core i5 2.5GHz, 8GB RAM) to finish the analysis of the program (with a size of 934 LoC). To conduct the analysis, we wrote a small driver program (about 60 LoC) which sets various configuration options of Joana, initiates the PDG construction, identifies and annotates the appropriate nodes in the PDG, and triggers the slicing-based information flow analysis.

We emphasize that while $\mathsf{EV}^*_{\vec{u}}[b]$ contains the ideal functionalities for public-key encryption and digital signatures that we use, and which hence, are part of the code analyzed by Joana, and also KeY (see below), these tools do *not* have to analyze the code of the untrusted libraries, namely network and cryptographic libraries (see also Figure 1). These libraries are subsumed by the (malicious) environment.

F. Proving the Conservatism Property with KeY

We have successfully used KeY to prove that

$$\mathsf{EV}^*_{\vec{u}}[b] \text{ is a conservative extension of } \mathsf{EV}_{\vec{u}}[b] \text{ for all } \vec{u}. \tag{7}$$

Therefore, together with (6), by our hybrid approach (Theorem 1), this implies (5), and hence, by the CVJ framework, we obtain the desired cryptographic privacy property (3).

At its core, as already mentioned in Section VI-D, verifying (7) involved proving that the voting machine calculates the result correctly. However, with Java being an imperative and object-oriented language one also has to make sure that if one component calls another one, then this other component does not have unexpected side effects on the calling component. Hence, often one has to prove that components are sufficiently separated. While KeY can prove this, in some places it was more efficient to make use of the PDG that Joana computed, from which the separation of certain components can easily be read off. This information directly implied that in order to verify (7) certain parts of the program were provably not relevant and could be ignored by KeY. So, interestingly, even in the "theorem proving part" of our analysis, we could make use of the information provided by the automated tool. However, clearly there is a limit to what Joana can do: in our case study, Joana certainly cannot prove (7) on its own; this, as mentioned, requires a theorem prover, such as KeY.

To give a feel for the kind of analysis performed in KeY, Fig. 2 displays one of the Java methods that had to be analyzed, which checks whether two arrays of equal length have equal integer arguments. The code to be analyzed needs to be annotated with specifications, i.e., pre- and post-conditions as well as loop invariants, which then need to be proven by KeY. The latter, in general, might again require some human interaction if the prover cannot verify the conditions automatically.

The part of the program that needs to be analyzed in order to prove (7) comprises 9 Java classes with 40 relevant methods in about 359 LoC. The methods were specified in 498 LoS (lines of specification), resulting in an overall LoS-to-LoC ratio of 1.39. The comprehensive specification process enabled an automated verification process (i.e., no user interaction was necessary) in KeY 2.4.0. It comprises 39 proofs, which take 17 minutes of computation time in total, with the longest proof taking 260 s, and 25 s on average (on an Intel Core 2 Duo at 2.66 GHz with 4 GB RAM). This resulted in a total of 200,843 rule applications, the longest proof requiring 72,109 rule applications. The entire specification and verification process took between two and three person weeks.

We note that KeY is able to prove noninterference properties itself [21], [38], by using the technique of self composition [15]. (Though, again, we emphasize that proving (7) does not involve proving noninterference properties.) Hence,

```
1  /*@ private normal_behaviour
2    @ requires r1.length == r2.length;
3    @ ensures \result == (\forall int i;
4    @     0 <= i && i < r1.length; r1[i] == r2[i]);
5    @ strictly_pure helper @*/
6  private static boolean equalResult(int[] r1, int[] r2) {
7      /*@ maintaining 0 <= j && r1.length == r2.length;
8        @ maintaining (\forall int i; 0 <= i && i < j;
9        @                              r1[i] == r2[i]);
10       @ assignable \strictly_nothing;
11       @ decreases r1.length - j;
12       @*/
13     for (int j= 0; j<r1.length; j++)
14         if (r1[j]!=r2[j]) return false;
15     return true;
16 }
```

Figure 2. JML annotated method to compare result vectors. JML annotations appear as comments with the special delimiter pair `/*@ @*/`. Lines 1ff. (before the method signature) state the contract. Line 2 imposes a precondition, while Line 3 defines a postcondition. Termination and absence of exceptions is included by default as stated by **`normal_behavior`**. The modifiers in Line 5 additionally require the method to not change the value of any location. Since the method contains an unbounded loop, we need to annotate that as well in order to prove the contract. A loop invariant is given in Lines 7ff. To prove termination, we also have a variant clause in Line 11.

in principle, we could have used KeY to verify property (6) directly without Joana (and the hybrid approach). However, given our experience with KeY and other theorem proving approaches, this would have been practically infeasible, both because of scalability problems and the overwhelming amount of human interaction that would have been necessary. At the very least, our hybrid approach tremendously simplified the analysis effort as it opened the way to use an automated tool.

VII. Related Work

Analysis based on type systems have been the predominant information flow analysis technique until recently. These approaches are usually sound, i.e., they are guaranteed to find any security violations. But they are incomplete w.r.t. sophisticated policies such as noninterference. This means that they tend to report *false positive* results. There exists a plethora of type systems for different programing languages using different extensions to narrow (but not close) the incompleteness gap. A widely used implementation of type-based analysis for sequential Java programs is the Jif compiler [52]. Other type systems for sequential Java include [5], [6], [7]. A related technique is explicit *dependency tracking* of variables. For example, [53], [22] present approaches—built on symbolic execution—in which for each variable a set of variables is stored on which its value at most depends.

Amtoft and Banerjee [13] were among the first to encode noninterference in a program logic and with that laid the foundation for sound and complete reasoning about information flows. Since classical Hoare logic cannot express noninterference, they employed a dedicated extension of it [54]. In [17], this line of research is continued, but using dynamic logic instead which can express such properties readily. In [21], [20], several practical improvements to this approach were presented and implemented in KeY. To the best of our knowledge, this is the only logic-based noninterference analysis for Java. However, previous experiments show that conducting a proof on a large system, such as the e-voting system in our case study, is not feasible with this approach alone.

None of the existing approaches for checking noninterference properties (for Java programs) appear to be applicable on their own to our e-voting case study. The hybrid approach proposed in this paper, and hence, the combination of automated and interactive tools, was key in tackling this already quite complex system.

An approach for code-level cryptographic verification similar to the CVJ framework approach has been presented in [27], but for the language F# and using refinement types (see [25] for a more detailed discussion).

VIII. Conclusion

In this paper, we have proposed what we call a hybrid approach for checking noninterference properties of Java program. This approach allows one to combine (fast) automated, but possibly imprecise analysis, with precise, but typically interactive tools, and hence, more time consuming analysis. By our approach, automated tools can be used as much as possible and only in places of a program where the automated tools fail, one resorts to more precise analysis.

We have successfully applied our approach to a non-trivial Java program, an e-voting system, for which we established noninterference. Together with the CVJ framework, our analysis implied strong cryptographic vote privacy. As explained in Section VI, the analysis of this system seems practically infeasible without the combination of different approaches into a hybrid approach: it appears that for our case study all fully automated tools which require almost no human input or interaction would fail; checking noninterference properties directly using (interactive) theorem proving approaches would be very complex and time consuming, and in fact, seems to be impractical with current analysis tools for Java.

The hybrid approach does not depend on specific tools and the basic idea should be applicable to other languages as well. Hence, it would be interesting to apply it for different languages—for example, functional languages for which verification is simpler compared to imperative and object-oriented languages—and using different tools. So far, we have formulated the hybrid approach for sequential programs. It would also be interesting future work to extend the approach to deal with concurrency.

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

References

[1] A. Sabelfeld and A. C. Myers, "Language-Based Information-Flow Security," *IEEE Journal on Selected Areas in Communications, special issue on Formal Methods for Security*, vol. 21, no. 1, pp. 5–19, 2003.

[2] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[3] D. Volpano, G. Smith, and C. Irvine, "A sound type system for secure flow analysis," *Journal of Computer Security*, vol. 4, no. 3, pp. 167–187, Dec. 1996. [Online]. Available: http://www.cs.nps.navy.mil/research/languages/papers/atsc/jcs.ps.Z

[4] D. M. Volpano and G. Smith, "Eliminating covert flows with minimum typings," in *10th Computer Security Foundations Workshop (CSFW '97), June 10-12, 1997, Rockport, Massachusetts, USA*, 1997, pp. 156–169.

[5] M. Strecker, "Formal analysis of an information flow type system for MicroJava," Technische Universität München, Tech. Rep., Jul. 2003.

[6] A. Banerjee and D. A. Naumann, "Stack-based access control and secure information flow," *J. Funct. Program.*, vol. 15, no. 2, pp. 131–177, 2005.

[7] G. Barthe, D. Pichardie, and T. Rezk, "A certified lightweight non-interference Java bytecode verifier," *Mathematical Structures in Computer Science*, vol. 23, no. 5, pp. 1032–1081, 2013.

[8] M. Alba-Castro, M. Alpuente, and S. Escobar, "Abstract Certification of Global Non-interference in Rewriting Logic," in *Formal Methods for Components and Objects - 8th International Symposium (FMCO 2009). Revised Selected Papers*, ser. Lecture Notes in Computer Science, F. S. de Boer, M. M. Bonsangue, S. Hallerstede, and M. Leuschel, Eds., vol. 6286. Springer, 2009, pp. 105–124.

[9] C. Hammer and G. Snelting, "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs," *Int. J. Inf. Sec.*, vol. 8, no. 6, pp. 399–422, 2009.

[10] A. C. Myers, S. Chong, N. Nystrom, L. Zheng, and S. Zdancewic, *Jif: Java Information Flow (software release)*, July 2001, http://www.cs.cornell.edu/jif/.

[11] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "Taj: effective taint analysis of web applications," *SIGPLAN Not.*, vol. 44, no. 6, pp. 87–97, 2009.

[12] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg, "Saving the world wide web from vulnerable javascript," ser. ISSTA '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/2001420.2001442

[13] T. Amtoft and A. Banerjee, "Information flow analysis in logical form," in *SAS*, 2004, pp. 100–115.

[14] A. Banerjee, D. A. Naumann, and S. Rosenberg, "Expressive declassification policies and modular static enforcement," *IEEE Symp. on Security and Privacy*, pp. 339–353, 2008.

[15] G. Barthe, P. R. D'Argenio, and T. Rezk, "Secure Information Flow by Self-Composition," in *17th IEEE Computer Security Foundations Workshop, (CSFW-17 2004)*. IEEE Computer Society, 2004, pp. 100–114.

[16] L. Beringer and M. Hofmann, "Secure information flow and program logics," in *20th IEEE Computer Security Foundations Symposium, CSF 2007, 6-8 July 2007, Venice, Italy*, 2007, pp. 233–248.

[17] Á. Darvas, R. Hähnle, and D. Sands, "A theorem proving approach to analysis of secure information flow," in *Proceedings, Security in Pervasive Computing*, ser. LNCS 3450, D. Hutter and M. Ullmann, Eds. Springer, 2005.

[18] K. R. M. Leino and R. Joshi, "A semantic approach to secure information flow," in *Mathematics of Program Construction, MPC'98, Marstrand, Sweden, June 15-17, 1998, Proceedings*, J. Jeuring, Ed., 1998, pp. 254–271.

[19] A. Nanevski, A. Banerjee, and D. Garg, "Verification of information flow and access control policies with dependent types," in *Security and Privacy (SP), 2011*, may 2011, pp. 165–179.

[20] C. Scheben, "Program-level specification and deductive verification of security properties," Ph.D. dissertation, Karlsruhe Institute of Technology, 2014, submitted.

[21] C. Scheben and P. H. Schmitt, "Verification of information flow properties of Java programs without approximations," in *Formal Verification of Object-Oriented Software*, ser. LNCS 7421. Springer, 2012, pp. 232–249.

[22] B. van Delft, "Abstraction, objects and information flow analysis," Master's thesis, Institute for Computing and Information Science, Radboud University Nijmegen, 2011.

[23] D. von Oheimb, "Formal methods in the security business: exotic flowers thriving in an expanding niche," in *14th International Symposium on Formal Methods, FM2006*, J. Misra, T. Nipkow, and E. Sekerinski, Eds., 2006, pp. 592–597.

[24] A. Nanevski, A. Banerjee, and D. Garg, "Dependent type theory for verification of information flow and access control policies," *ACM Trans. Program. Lang. Syst*, vol. 35, no. 2, p. 6, 2013.

[25] R. Küsters, T. Truderung, and J. Graf, "A Framework for the Cryptographic Verification of Java-like Programs," in *25th IEEE Computer Security Foundations Symposium (CSF 2012)*. IEEE Computer Society, 2012, pp. 198–212.

[26] G. Klein and T. Nipkow, "A Machine-Checked Model for a Java-Like Language, Virtual Machine, and Compiler," *ACM Trans. Program. Lang. Syst.*, vol. 28, no. 4, pp. 619–695, 2006.

[27] C. Fournet, M. Kohlweiss, and P.-Y. Strub, "Modular code-based cryptographic verification," in *Proceedings of the 18th*

ACM Conference on Computer and Communications Security (CCS 2011), Y. Chen, G. Danezis, and V. Shmatikov, Eds. ACM, 2011, pp. 341–350.

[28] G. Barthe, C. Fournet, B. Grégoire, P.-Y. Strub, N. Swamy, and S. Zanella Béguelin, "Probabilistic relational verification for cryptographic implementations," *SIGPLAN Not.*, vol. 49, no. 1, pp. 193–205, Jan. 2014. [Online]. Available: http://doi.acm.org/10.1145/2578855.2535847

[29] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, P. Strub, and S. Z. Béguelin, "Proving the TLS Handshake Secure (As It Is)," in *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Proceedings, Part II*, ser. Lecture Notes in Computer Science, J. A. Garay and R. Gennaro, Eds., vol. 8617. Springer, 2014, pp. 235–255.

[30] R. Küsters, E. Scapin, T. Truderung, and J. Graf, "Extending and Applying a Framework for the Cryptographic Verification of Java Programs," in *Principles of Security and Trust - Third International Conference, POST 2014*, ser. Lecture Notes in Computer Science, M. Abadi and S. Kremer, Eds., vol. 8414. Springer, 2014, pp. 220–239, a full version is available at http://eprint.iacr.org/2014/038.

[31] M. Aizatulin, A. D. Gordon, and J. Jürjens, "Computational verification of C protocol implementations by symbolic execution," in *ACM Conference on Computer and Communications Security*, T. Yu, G. Danezis, and V. D. Gligor, Eds. ACM, 2012, pp. 712–723.

[32] R. Canetti, "Universally Composable Security: A New Paradigm for Cryptographic Protocols," in *Proceedings of the 42nd Annual Symposium on Foundations of Computer Science (FOCS 2001)*. IEEE Computer Society, 2001, pp. 136–145.

[33] B. Pfitzmann and M. Waidner, "A Model for Asynchronous Reactive Systems and its Application to Secure Message Transmission," in *IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2001, pp. 184–201.

[34] R. Küsters, "Simulation-Based Security with Inexhaustible Interactive Turing Machines," in *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW-19 2006)*. IEEE Computer Society, 2006, pp. 309–320, see http://eprint.iacr.org/2013/025/ for a full and revised version.

[35] W. Ahrendt, B. Beckert, D. Bruns, R. Bubel, C. Gladisch, S. Grebing, R. Hähnle, M. Hentschel, M. Herda, V. Klebanov, W. Mostowski, C. Scheben, P. H. Schmitt, and M. Ulbrich, "The KeY platform for verification and analysis of Java programs," in *Verified Software: Theories, Tools, and Experiments (VSTTE 2014)*, ser. Lecture Notes in Computer Science, D. Giannakopoulou and D. Kroening, Eds., no. 8471. Springer-Verlag, 2014, pp. 1–17. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-319-12154-3_4

[36] R. Küsters, T.Truderung, B. Beckert, D. Bruns, M. Kirsten, and M. Mohr, "Code and Additional Verification Data for a Case Study: Verification of an E-voting System using the Hybrid Approach," 2015, https://infsec.uni-trier.de/download/HybridApproachEVotingCaseStudyZIP-2015/EVotingMachine.zip.

[37] J. A. Goguen and J. Meseguer, "Security Policies and Security Models," in *Proceedings of IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[38] B. Beckert, D. Bruns, V. Klebanov, C. Scheben, P. H. Schmitt, and M. Ulbrich, "Information flow in object-oriented software," in *Logic-Based Program Synthesis and Transformation, LOPSTR 2013*, ser. Lecture Notes in Computer Science, G. Gupta and R. Peña, Eds., no. 8901. Springer, 2014, pp. 19–37.

[39] J. Graf, M. Hecker, and M. 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)*, ser. Lecture Notes in Informatics (LNI) 215. Springer Berlin / Heidelberg, Feb. 2013.

[40] D. Wasserrab, "From Formal Semantics to Verified Slicing - A Modular Framework with Applications in Language Based Security," Ph.D. dissertation, Karlsruher Institut für Technologie, Fakultät für Informatik, Oct. 2010. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000020678

[41] D. Wasserrab and D. Lohner, "Proving Information Flow Noninterference by Reusing a Machine-Checked Correctness Proof for Slicing," in *6th International Verification Workshop - VERIFY-2010*, Jul. 2010.

[42] D. Wasserrab, D. Lohner, and G. Snelting, "On PDG-Based Noninterference and its Modular Proof," in *Proceedings of the 4th Workshop on Programming Languages and Analysis for Security*. ACM, Jun. 2009, pp. 31–44. [Online]. Available: http://pp.info.uni-karlsruhe.de/uploads/publikationen/wasserrab09plas.pdf

[43] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt, "The KeY tool," *Software and System Modeling*, vol. 4, pp. 32–54, 2005.

[44] B. Beckert, R. Hähnle, and P. H. Schmitt, Eds., *Verification of Object-Oriented Software: The KeY Approach*, ser. Lecture Notes in Computer Science. Springer, 2007, no. 4334.

[45] B. Beckert, "A dynamic logic for Java Card," in *Proceedings, 2nd ECOOP Workshop on Formal Techniques for Java Programs, Cannes, France*, 2000, pp. 111–119.

[46] G. T. Leavens, A. L. Baker, and C. Ruby, "JML: a Java Modeling Language," in *Formal Underpinnings of Java Workshop (at OOPSLA '98)*, Oct. 1998.

[47] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, D. M. Zimmerman, and W. Dietl, *JML Reference Manual*, May 31 2013, draft revision 2344. [Online]. Available: http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_toc.html

[48] M. Hentschel, S. Käsdorf, R. Hähnle, and R. Bubel, "An interactive verification tool meets an IDE," in *Proceedings of the 11th International Conference on Integrated Formal Methods*, ser. LNCS, G. Z. Elvira Albert, Emil Sekerinski, Ed. Springer, Sep. 2014, pp. 55–70.

[49] M. J. Fischer and R. E. Ladner, "Propositional dynamic logic of regular programs," *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 194–211, Apr. 1979.

[50] D. Harel, "Dynamic logic," in *Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic*, D. Gabbay and F. Guenther, Eds. Dordrecht: D. Reidel Publishing Co., 1984, pp. 497–604.

[51] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, vol. 12, no. 10, 1969.

[52] A. C. Myers, "JFlow: Practical mostly-static information flow control," in *The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '99)*. New York: Association for Computing Machinery, Jan. 1999, pp. 228–241.

[53] R. Bubel, R. Hähnle, and B. Weiß, "Abstract interpretation of symbolic execution with explicit state updates," in *Post Conf. Proc. 6th International Symposium on Formal Methods for Components and Objects (FMCO)*, ser. Lecture Notes in Computer Science, F. de Boer, M. M. Bonsangue, and E. Madelaine, Eds., vol. 5751. Springer-Verlag, 2009, pp. 247–277.

[54] T. Amtoft and A. Banerjee, "Verification condition generation for conditional information flow," in *Proceedings of the 2007 ACM workshop on Formal methods in security engineering*, P. Ning, V. Atluri, V. D. Gligor, and H. Mantel, Eds. New York, NY, USA: ACM, 2007, pp. 2–11.

[55] R. Küsters, T. Truderung, and J. Graf, "A Framework for the Cryptographic Verification of Java-like Programs," Cryptology ePrint Archive, Report 2012/153, 2012, http://eprint.iacr.org/2012/153.

Appendix A. Proof of Theorem 1

To prove Theorem 1, we first need to introduce some notions.

Let us first recall that a *configuration* of Jinja+, as defined in [25] (see also Appendix B), is of the form $\langle e, s\rangle$, where e is a Jinja+ expression and s is a state. A *state* is a pair $\langle h, l\rangle$ of a heap and a store. A *store* is a map from variable names to values. A *heap* is a map from references (addresses) to object instances. An *object instance* is a pair consisting of a class name and a *field table*, and a field table is a map from field names to values. A value can be either a reference or a value of a primitive type.

By $E[\cdot]$ we will denote an *expression context*, that is a Jinja+ expression with a hole. $E[e]$ will denote the expression obtained by substituting replacing the hole in E with the expression e.

Structure of states in a run. One particular type of expressions is a *block expression* of the form

$$\{v : t;\ e\}_C \quad \text{or} \quad \{v : t;\ v := \mathsf{val};\ e\}_C,$$

where v is a local variable (whose scope is this block) of type t and, in the second variant, with value val, e is an expression (e can access the local variable v), and C is a class name (denoting that the block originates from the code of class C). Such block expressions are introduced by the rule for the method call (33), where the block is, essentially, the method body and C indicates the origin of the block expression (meaning that the method is defined in class C).

In general, an expression may contain many blocks as its subexpression. However, when we study expressions that occur in actual runs of Jinja+, it turns out that they have a simpler form, where all blocks are located on one path. Formally, let

$$c_0 = \langle e_0, \langle h_0, l_0\rangle\rangle \xrightarrow{\ell} \langle e_1, \langle h_1, l_1\rangle\rangle \xrightarrow{\ell} \cdots$$

be a run (with the initial configuration c_0). By the definition of the initial configuration [25], [55], h_0 is empty and l_0 bounds the static variables of the program to their initial values (and no other variables). By inspecting the rules of Jinja+ [55] (see Appendix B), one can see that, for every $i = 0, 1, \ldots,$

- l_i bounds only static variables,
- for every subexpression e of e_i, either e contains no block as its subexpression or e is of the form $E[b]$, where E contains no block and b is a block expression. It means that e can contain, directly, at most one block (although b can contain further, nested blocks).

The reasoning given below will apply to expressions originating from runs of Jinja+ systems and, therefore, having the above form.

We also can observe, again inspecting the rules of Jinja+, that whenever a configuration (e_1, s_1) is reduced in one step to (e_2, s_2), the reduction happens within the innermost block expression (if any). Formally, e_1 is of the form $E[e_1']$, $e_2 = E[e_2']$, (e_1', s_1) reduces in one step to (e_2', s_2) and, moreover, one of the following cases happens:

- e_1' does not have block expressions,
- e_1' is of the form $\{V : T;\ V := \texttt{Val}\ v;\ \texttt{Val}\ u\}_D$ or $\{V : T;\ \texttt{Val}\ u\}_D$ and e_1 reduces (in one step) to $\texttt{Val}\ u$ (by (34) or (34)).

The second case describes the step when a method executions is finished and returns value u.

Structure of a conservative extension. Let P' be a conservative extension of P. By the definition, P' is of the form $P^* \cdot M$, where P^* is obtained from P by adding statements of the form (1) and (2). Formally, it means that some subexpressions e of the program P are replaced by $(e^*; e)$, where e^* is of the form (1) or (2).

We will call an block expression $\{\cdots\}_D$ with D defined in M, an *M-block*.

Pruning. We now define the pruning operations for expressions originating from a run of P'. Intuitively, the pruning of an expression e is obtained from e by leaving only those parts of e that have originated from a particular component of the system (such as the original program P).

Let e be such an expression. We define the *P-pruning of e*, written $\Delta_P(e)$, to be the expression obtained from e

by removing all subexpressions of e of the form (1), (2) and all M-blocks. Formally, prunning is defined recursively as follows. For an expression of the form $e = (e_1; e_2)$, where e_1 is either of the form (1) or (2) or an M-block, we define $\Delta_P(e)$ as $\Delta_P(e_2)$. Otherwise, prunning is applied recursively to direct subexpression of e, i.e. for e of the form $F[e_1,\ldots,e_n]$, where F is the constructor (function symbol in the head) of the expression e and $e_1,\ldots,e_n$ is the (possibly empty) list of direct subexpressions of e, we define $\Delta_P(e)$ as $F[\Delta_P(e_1),\ldots,\Delta_P(e_n)]$. In particular, $\Delta_P(e) = e$, if e is a value or a variable.

As we can see, P-pruning removes the sub-expressions that originated from P^* or M but not from P.

We also define P-pruning $\Delta_P(l)$ for a store l, by removing from l all (static) variables declared in classes of M (and hence only leaving those variables that are declared in the original program P).

Given a configuration $c = \langle e, \langle h, l\rangle\rangle$, we will write $\Delta_P(c)$ for $\langle \Delta_P(e), \langle h, \Delta_P(l)\rangle\rangle$.

Analogously, we define M-pruning, $\Delta_M(e)$, as the set of all subexpressions of e which are M-blocks. M-pruning $\Delta_M(l)$ for a store l is obtained by removing from l all (static) variables declared in classes of P (and leaving only those static variables that are defined in M).

Reference closure. For a set of references R and a heap h, we define the *closure of R under h*, as the smallest superset R' of R such that if $r \in R'$ and r is of some class D with a field m of a reference type, then $h(r).m \in R'$.

Equality up to reference renaming. Let f be a bijection on the set of references. Let R be a set of references and h_1, h_2 be heaps. We write $h_1 \sim_{f,R} h_2$ if for all $r \in R$

- r and $r' = f(r)$ are of the same class D and
- if m is a field of D of a reference type, then $f(h_1(r).m) = h_2(r').m$ and $h(r).m$ is in R.

Two configurations $c_1 = \langle e_1, \langle h_1, l_1\rangle\rangle$ and $c_2 = \langle e_2, \langle h_2, l_2\rangle\rangle$ are equal up to reference renaming, written $c_1 \sim c_2$, if there exists a bijection f on the set of references such that:

- $e_2 = f(e_1)$ (here the function f, originally defined for references, is extended by structural isomorphism to all expressions)
- $l_2 = f(l_1)$ (where $f(l_1)$ is, again, extended by structural isomorphism to stores).
- for the closure R of the set of references occurring in e_1 and in l_1 under h, we have that $h \sim_{f,R} h'$.

To make it explicit which bijection f is used in the above definition, we can also write $c_1 \sim_f c_2$.

Disjoint states. Let $c = \langle e, \langle h, l\rangle\rangle$ be a configuration of P'. We say that P and M have *disjoint states* in c, if R_P and R_M are disjoint, where R_P is the closure of the set of references occurring in $\Delta_P(e)$ and $\Delta_P(l)$ under h and, similarly, R_M and is the closure of the set of references occurring in $\Delta_M(e)$ and $\Delta_M(l)$ under h.

It is easy to show that, because only values of primitive types are exchanged between P and M, the states of these components are separate in all configurations of P':

Lemma 1. *Let c be a configuration reachable from the initial configuration of $P'[\vec{a}]$, for some $\vec{a}$. Then P and M have disjoint states in c.*

Proof: By the definition of the initial configuration and by the definition of P', it is easy to see that in the initial configuration $c_0 = \langle e_0, h_0, l_0\rangle$ of $P'[\vec{a}]$, P and M have disjoint states. In particular, e_0 does not contain M-blocks and, therefore, the set R_M of references, as defined above, for c_0, is empty (in the initial configuration, variables of reference types in l_0 can be only initialized with `null`).

Now, we can show, considering all rules of Jinja+, that the property of P and M having disjoint states is preserved by the application of any rule.

Note that the original code of P never writes to (static) variables of M and, analogously, the code of m never writes to variables of P. So the only possible rules that could break this property are:

- the rule for method call (33) which creates an M-block and passes some values to this block,
- the return rules (34) and (35) which reduce an M-block to its final value,
- the expression propagation rules (57) and (58) that propagate an exception from an M-block (deleting this block).

For the first two cases, we note that both the arguments of a method call and the return value are of primitive types, and therefore these rules do not break the property of disjoint states. One can also see that, by the definition of a conservative extension, the last case cannot happen (exceptions are always caught inside M). ■

We will write $c \xrightarrow{M} c'$, if $c \xrightarrow{D} c'$ with D defined in M. It means that c reduces to c' in one step which is taken in the code of class $D \in M$.

If two components have disjoint states, then an action taken in one of the does not change the state of the second, unless the action is method call or return from a method. In particular, actions taken within M do not have any effect of the state of P, as stated in the following lemma.

Lemma 2. *Let c be a configuration of $P'[\vec{a}]$, for some $\vec{a}$, in which P and M have disjoint states. Suppose that $c \xrightarrow{M} c'$, where this reduction is not obtained by the return rules (34), (35). Then the state of P is not changed by this step, i.e. $\Delta_P(c) = \Delta_P(c')$.*

Proof: The proof proceeds easily by considering all rules of Jinja+, and by induction on the size of the reduced expression.

We only note here that, in particular, $\Delta_P(e) = \Delta_P(e')$, where $c = \langle e,h,l\rangle$ and $c' = \langle e',h',l'\rangle$, as the reduction takes place inside an M-block. Also, $\Delta_P(l) = \Delta_P(l')$, because (a) the code of M (code inside an M-block) does not write directly to static methods of P and (b) P and M have disjoint states (no reference reachable from l is reachable from M). ∎

Lemma 3. *Let c be a configuration of $P[\vec{a}]$, for some $\vec{a}$, and $\hat{c}$ be a configuration of $P'[\vec{a}]$ such that $\hat{c}$ contains no M-blocks and $\Delta_P(\hat{c}) \sim c$. Suppose that $c \to c'$ (i.e. c reduces to c' in one step).*

Then $\hat{c} \xrightarrow{*} \hat{c}'$ (i.e. $\hat{c}$ reduces, possibly in more than one step, to $\hat{c}'$) such that $\hat{c}'$ contains no M-blocks and $\Delta_P(\hat{c}') \sim c'$.

Proof: Let $c = \langle e,h,l\rangle$ and $\hat{c} = \langle \hat{e},\hat{h},\hat{l}\rangle$. The proof proceeds by induction on the size of the terms in the configurations c and $\hat{c}$. We consider, case by case, different forms of the expression e which triggers different rules in the reduction step of c. We present here only some chosen cases.

The interesting case is the when $\hat{e} = (\hat{e}_1;\hat{e}_2)$, where $\hat{e}_1$ is of the form (1) or (2). We will consider here the latter case, i.e. $\hat{e}_1 = (v = C.f(d_1,\ldots,d_k))$ (as the more interesting one; we can deal with the former case in a very similar manner). Note that in this case $\hat{e}_2$ contains no M-blocks and, by the definition of pruning,

$$\Delta_P(\langle \hat{e}_2,\hat{h},\hat{l}\rangle) \sim c. \tag{8}$$

Because d_i (for $i \in \{1,\ldots,k\}$) are without side effects, $\hat{c}$ reduces (in some number of steps) to

$$\langle (v := C.f(v_1,\ldots,v_k);\ \hat{e}_2),\ \hat{h},\hat{l}\rangle$$

where v_i are some values (without changing the state $\hat{h},\hat{l}$). Further, this configuration is reduced, again without changing the state, by the method call rule (33), which introduces an M-block, to a configuration of the form

$$\hat{c}' \ = \ \langle (v := \{\ldots\}_D;\ \hat{e}_2),\ \hat{h},\hat{l}\rangle,$$

where D is a class defined in M. This, configuration, in turn—because we assume that methods of M terminate and do not propagate exceptions—is reduced in some finite number of steps labeled by M to a configuration of the form

$$\hat{c}'' \ = \ \langle (v := \{\ldots;\ \texttt{Val}\ u\}_D;\ \hat{e}_2),\ \hat{h}',\ \hat{l}'\rangle$$

and further, by a return rule ((34) or (35)) to

$$\langle (v := \texttt{Val}\ u;\ \hat{e}_2),\ \hat{h}',\ \hat{l}'\rangle.$$

By the condition of a conservative extension, we know that this assumption does not change the state of the stystem (the variable v already contains the value $\texttt{Val}\ u$). Therefore, the above configuration reduces, by rule (29) and then (36), to

$$\hat{c}''' = \langle \hat{e}_2,\ \hat{h}',\ \hat{l}'\rangle.$$

Now, recall that the configuration $\hat{c}''$ was obtained from $\hat{c}'$ using only M-steps. Therefore by Lemma 2, $\Delta_P(\hat{c}') = \Delta_P(\hat{c}'')$. This, together with (8), implies that $\Delta_P(\hat{c}''') \sim c$. Recall also that $\hat{e}_2$, and hence also $\hat{c}'''$, contains no M-blocks.

We can now use the inductive hypothesis to conclude that $\hat{c}'''$ further reduces (in some number of steps) to $\hat{c}''''$ such that $\hat{c}''''$ contains no M-blocks and $\Delta_P(c'''') \sim c'$. This completes the proof for this case.

The remaining cases are much simpler. Let us now consider the case representative of subexpression reduction rules, where $e = \texttt{if}\ (e_1)\ e_2\ \texttt{else}\ e_3$ gets reduced to $e' = \texttt{if}\ (e_1')\ e_2\ \texttt{else}\ e_3$ with state h',l', because $\langle e_1,h,l\rangle$ reduces to $\langle e_1',h',l'\rangle$. In this case, $\hat{e} = \texttt{if}\ (\hat{e}_1)\ \hat{e}_2\ \texttt{else}\ \hat{e}_3$ and $\Delta_P(\hat{e}_i) \sim e_i$ for $i \in \{1,\ldots,3\}$. By the inductive hypothesis $\hat{e}_1$ reduces (in some number of steps) to some $\hat{e}_1'$ changing the state from $\hat{h},\hat{l}$ to $\hat{h}',\hat{l}'$, where $\hat{e}_1'$ contains no M-blocks and $\Delta\langle \hat{e}_1',\hat{h}',\hat{l}'\rangle \sim \langle e_1',h',l'\rangle$. Therefore, $\hat{c}$ reduces to $\hat{c}' = \langle \hat{e}',\hat{h}',\hat{l}'\rangle$ with $\hat{e}' = \texttt{if}\ (\hat{e}_1')\ \hat{e}_2\ \texttt{else}\ \hat{e}_3$. We conlude the proof for this case observing that $\hat{c}'$ contains no M-blocks and $\Delta_P(\hat{c}') \sim c'$. ∎

Similarly, we can prove the following result, concerning the final configuration:

Lemma 4. *Let c be the final configuration of $P[\vec{a}]$, for some $\vec{a}$, and c' be a configuration of $P'[\vec{a}]$ such that c' contains no M-blocks and $\Delta_P(c') \sim c$.*

Then either c' is final or $c' \xrightarrow{*} c''$ (i.e. c' reduces, possibly in more than one step, to c'') such that c'' is final, contains no M-blocks, and $\Delta_P(c'') \sim c$.

Proof of Theorem 1. Suppose that the program $P'[\vec{x}]$ is noninterferent, whereas the original program $P[\vec{x}]$ is not. It means that there exist $\vec{a}_1$ and $\vec{a}_2$ such that $P[\vec{a}_1]$ and $P[\vec{a}_2]$ terminate with final values of low variables $\vec{y}$, respectively, $\vec{b}_1$ and $\vec{b}_2$, where $\vec{b}_1 \neq \vec{b}_2$.

Let $c_0,c_1,\ldots,c_n$ be the run of $P[\vec{a}_1]$ and let c_0' be the initial state of $P'[\vec{a}_1]$. We can use Lemma 3 to obtain $c_1',\ldots,c_n'$ such that $c_0' \xrightarrow{*} c_1' \xrightarrow{*} \cdots \xrightarrow{*} c_n'$ and, for each $i \in \{0,\ldots,n\}$

$$c_i' \text{ contains no } M\text{-blocks and } \Delta(c_i') \sim c_i \tag{9}$$

Indeed, it is easy to see, by the construction of P', that (9) holds for $i = 0$ (that is for the initial states). Then, for $i > 0$, we easily obtain (9) by induction, using Lemma 3.

Now, since c_n is a final state (i.e. it cannot be further reduced), we know by Lemma 4 that c_n' reduces in zero or more steps to c_n'' such that c_n'' is final, contains no M-blocks and $\Delta_P(c_n'') \sim c_n$. The latter means, in particular, that the values of the low variables $\vec{y}$ are the same in c_n and in c_n''. Therefore $P[\vec{a}_1]$ and $P'[\vec{a}_1]$ terminate with the same values $\vec{b}_1$ of low variables $\vec{y}$.

In the same way we show that $P[\vec{a}_2]$ and $P'[\vec{a}_2]$ terminate with the same values $\vec{b}_2$ of low variables $\vec{y}$.

Recall that we have assumed that $\vec{b}_1 \neq \vec{b}_2$. This, however, means—contrary to what we have assumed—that the pro-

gram $P'[\vec{x}]$ is not noninterferent. Hence we have reached a contradiction and proven that $P[\vec{x}]$ is noninterferent.

APPENDIX B. JINJA+

As mentioned before, Jinja+ is based on *Jinja* [26] and extends this language with some additional features. We first shortly recall the language Jinja and then the extented language Jinja+.

A. Jinja

Syntax. Expressions in Jinja are constructed recursively and include: (a) creation of a new object, (b) casting, (c) literal values (constants) of types boolean and int, (d) `null`, (e) binary operations, (f) variable access and variable assignment, (g) field access and field assignment, (h) method call, (i) blocks with locally declared variables, (j) sequential composition, (k) conditional expressions, (l) while loop, (m) exception throwing and catching.

A *program* or a *system* is a set of class declarations. A *class declaration* consists of the name of the class and the class itself. A *class* consists of the name of its direct superclass (optionally), a list of field declarations, and a list of method declarations, where we require that different fields and methods have different names. A *field declaration* consists of a type and a field name. A *method declaration* consists of the method name, the formal parameter names and types, the result type, and an expression (the method body). Note that there is no `return` statement, as a method body is an expression; the value of such an expression is returned by the method.

Jinja comes equipped with a type system and a notion of well-typed programs. We consider only well-typed programs.

Semantics. Following [26], we briefly sketch the small-step semantics of Jinja.

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

The small-step semantics of Jinja is given by the set of rules of the form $P \vdash \langle e,s\rangle \to \langle e',s'\rangle$, describing a single step of the program execution (reduction of an expression), given in Figures 3, 4, 5, and 6. We will call $\langle e,s\rangle$ ($\langle e',s'\rangle$) a *configuration*. In this notation, 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'.

Following [25], [55], reduction rules are labeled. 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 empty 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 (17)–(19)).

To define labeling of transitions, labels are also added to blocks that are obtained from the method call rule (33) (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.

Subexpression reduction rules (Figure 3) describe the order in which subexpressions are evaluated. *Expression reduction* rules (Figure 4) are applied when the subexpressions are sufficiently reduced. *Exceptional reduction* and *exception propagation* rules (Figure 5 and 6) 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. Jinja+

Jinja+ extends Jinja with: (a) the primitive type `byte` with natural conversions from and to `int`, (b) arrays, (c) `abort` primitive, (d) static fields (with the restriction that they can be initialized by literals only), (e) static methods, (f) access modifier for classes, fields, and methods (such as `private`, `protected`, and `public`), (g) final classes (classes that cannot be extended), (h) the `throws` clause of a method declaration (that declare which exceptions can be thrown by a method). (i) 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*.

More details on these extensions can be found in [55]. Here we only discuss some of them.

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

Static methods. 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

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle \texttt{Cast}\ C\ e, s\rangle \xrightarrow{\ell} \langle \texttt{Cast}\ C\ e', s'\rangle} \tag{10}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle V := e, s\rangle \xrightarrow{\ell} \langle V := e', s'\rangle} \tag{11}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle e.F\{D\}, s\rangle \xrightarrow{\ell} \langle e'.F\{D\}, s'\rangle} \tag{12}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle e.F\{D\} := e_2, s\rangle \xrightarrow{\ell} \langle e'.F\{D\} := e_2, s'\rangle} \tag{13}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle \texttt{Val}\ v.F\{D\} := e, s\rangle \xrightarrow{\ell} \langle \texttt{Val}\ v.F\{D\} := e', s'\rangle} \tag{14}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle e \ll bop \gg e_2, s\rangle \xrightarrow{\ell} \langle e' \ll bop \gg e_2, s'\rangle} \tag{15}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle \texttt{Val}\ v_1 \ll bop \gg e, s\rangle \xrightarrow{\ell} \langle \texttt{Val}\ v_1 \ll bop \gg e', s'\rangle} \tag{16}$$

$$\frac{P \vdash \langle e, (h, l(V := None))\rangle \xrightarrow{\ell} \langle e', (h', l')\rangle \quad l'\ V = None \quad \neg\ assigned\ V\ e}{P \vdash \langle \{V : T; e\}_D, (h, l)\rangle \xrightarrow{f(l,D)} \langle \{V : T; e'\}_D, (h', l'(V := l\ V))\rangle} \tag{17}$$

$$\frac{P \vdash \langle e, (h, l(V := None))\rangle \xrightarrow{\ell} \langle e', (h', l')\rangle \quad l'\ V = v \quad \neg\ assigned\ V\ e}{P \vdash \langle \{V : T; e\}_D, (h, l)\rangle \xrightarrow{f(l,D)} \langle \{V : T;\ V := \texttt{Val}\ v; e'\}_D, (h', l'(V := l\ V))\rangle} \tag{18}$$

$$\frac{P \vdash \langle e, (h, l(V \mapsto v))\rangle \xrightarrow{\ell} \langle e', (h', l')\rangle \quad l'\ V = v'}{P \vdash \langle \{V : T; V := \texttt{Val}\ v; e\}_D, (h, l)\rangle \xrightarrow{f(l,D)} \langle \{V : T;\ V := \texttt{Val}\ v'; e'\}_D, (h', l'(V := l\ V))\rangle} \tag{19}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle e.M(es), s\rangle \xrightarrow{\ell} \langle e'.M(es), s'\rangle} \tag{20}$$

$$\frac{P \vdash \langle es, s\rangle\ [\xrightarrow{\ell}]\ \langle es', s'\rangle}{P \vdash \langle \texttt{Val}\ v.M(es), s\rangle \xrightarrow{\ell} \langle \texttt{Val}\ v.M(es'), s'\rangle} \tag{21}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle e; e_2, s\rangle \xrightarrow{\ell} \langle e'; e_2, s'\rangle} \tag{22}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle \texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2, s\rangle \to \langle \texttt{if}\ (e')\ e_1\ \texttt{else}\ e_2, s'\rangle} \tag{23}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle e \cdot es, s\rangle\ [\xrightarrow{\ell}]\ \langle e' \cdot es, s'\rangle} \qquad \frac{P \vdash \langle es, s\rangle\ [\xrightarrow{\ell}]\ \langle es', s'\rangle}{P \vdash \langle \texttt{Val}\ v \cdot es, s\rangle\ [\xrightarrow{\ell}]\ \langle \texttt{Val}\ v \cdot es', s'\rangle} \tag{24}$$

Figure 3. Subexpression reduction rules. We define $f(l,D) = D$, if $l = -$; otherwise $f(l,D) = l$.

$$\frac{\textit{new-Addr } h = a \quad P \vdash C \textit{ has-fields } FDTs}{P \vdash \langle \texttt{new } C, (h,l)\rangle \xrightarrow{-} \langle \textit{addr } a, (h(a \mapsto\ (C,\ \textit{init-fields } FDTs)), l)\rangle} \tag{25}$$

$$\frac{hp\ s\ a = (D, f_s) \quad P \vdash D \preceq^*\ C}{P \vdash \langle \texttt{Cast } C\ (\textit{addr } a), s\rangle \xrightarrow{-} \langle \textit{addr } a, s\rangle} \tag{26}$$

$$P \vdash \langle \texttt{Cast } C\ \textit{null}, s\rangle \xrightarrow{-} \langle \textit{null}, s\rangle \tag{27}$$

$$\frac{lcl\ s\ V = v}{P \vdash \langle \texttt{Var } V, s\rangle \xrightarrow{-} \langle \texttt{Val } v, s\rangle} \tag{28}$$

$$P \vdash \langle V := \texttt{Val } v, (h,l)\rangle \xrightarrow{-} \langle \textit{unit}, (h, l(V \mapsto\ v))\rangle \tag{29}$$

$$\frac{\textit{binop } (bop, v_1, v_2) = v}{P \vdash \langle \texttt{Val } v_1 \ll bop \gg \texttt{Val } v_2, s\rangle \xrightarrow{-} \langle \texttt{Val } v, s\rangle} \tag{30}$$

$$\frac{hp\ s\ a = (C, fs) \quad fs(F,D) = v}{P \vdash \langle \textit{addr } a.F\{D\}, s\rangle \xrightarrow{-} \langle \texttt{Val } v, s\rangle} \tag{31}$$

$$\frac{h\ a = (C, fs)}{P \vdash \langle \textit{addr } a.F\{D\} := \texttt{Val } v, (h,l)\rangle \xrightarrow{-} \langle \textit{unit}, (h(a\ \mapsto\ (C, fs((F,D)\ \mapsto\ v))), l)\rangle} \tag{32}$$

$$\frac{hp\ s\ a = (C, fs) \quad P \vdash\ C \textit{ sees } M:\ Ts \to T = (pns,\ body) \textit{ in } D \quad |vs| = |pns| \quad |Ts| = |pns|}{P \vdash \langle \textit{addr } a.M(\textit{map } \texttt{Val } vs), s\rangle \xrightarrow{-} \langle \textit{blocks}_D(\textit{this} \cdot pns,\ \textit{Class } D \cdot Ts,\ \textit{Addr } a \cdot vs,\ body), s\rangle} \tag{33}$$

$$P \vdash \langle \{V : T;\ V := \texttt{Val } v;\ \texttt{Val } u\}_D, s\rangle \xrightarrow{D} \langle \texttt{Val } u, s\rangle \tag{34}$$

$$P \vdash \langle \{V : T;\ \texttt{Val } u\}_D, s\rangle \xrightarrow{D} \langle \texttt{Val } u, s\rangle \tag{35}$$

$$P \vdash \langle \texttt{Val } v;\ e_2, s\rangle \xrightarrow{-} \langle e_2, s\rangle \tag{36}$$

$$P \vdash \langle \texttt{if}(\textit{true})\ e_1\ \texttt{else}\ e_2, s\rangle \xrightarrow{-} \langle e_1, s\rangle \tag{37}$$

$$P \vdash \langle \texttt{if}(\textit{false})\ e_1\ \texttt{else}\ e_2, s\rangle \xrightarrow{-} \langle e_2, s\rangle \tag{38}$$

$$P \vdash \langle \texttt{while}(b)\ c, s\rangle \xrightarrow{-} \langle \texttt{if}(b)\ (c;\ \texttt{while}(b)\ c)\ \texttt{else}\ \textit{unit}, s\rangle \tag{39}$$

Figure 4. Expression reduction rules. In the rule for method invocation, the required nested block structure is built with the help of the auxiliary function *blocks* defined as follows: $\textit{blocks}_C([],[],[],e) = e$ and $\textit{blocks}_C(V \cdot Vs, T \cdot Ts, v \cdot vs, e) = \{V : T;\ V := v;\ \textit{blocks}(Vs, Ts, vs, e)\}_C$, (where $\cdot$ is the list constructor and $[]$ denotes the empty list).

$$\frac{hp\ s\ a\ =\ (D,fs)\quad \neg\, P \vdash D \preceq^* C}{P \vdash \langle \texttt{Cast}\ C(addr\ a), s\rangle \xrightarrow{-} \langle \texttt{THROW ClassCastException},\ s\rangle} \tag{40}$$

$$P \vdash \langle null.F\{D\}, s\rangle \xrightarrow{-} \langle \texttt{THROW NullPointerException},\ s\rangle \tag{41}$$

$$P \vdash \langle null.F\{D\} := \texttt{Val}\ v, s\rangle \xrightarrow{-} \langle \texttt{THROW NullPointerException},\ s\rangle \tag{42}$$

$$P \vdash \langle null.M(map\ \texttt{Val}\ vs), s\rangle \xrightarrow{-} \langle \texttt{THROW NullPointerException},\ s\rangle \tag{43}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle \texttt{throw}\ e, s\rangle \xrightarrow{\ell} \langle \texttt{throw}\ e', s'\rangle} \tag{44}$$

$$P \vdash \langle \texttt{throw}\ null, s\rangle \xrightarrow{-} \langle \texttt{THROW NullPointerException}, s\rangle \tag{45}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle \texttt{try}\ e\ \texttt{catch}\ (C\ V)\ e_2, s\rangle \xrightarrow{\ell} \langle \texttt{try}\ e'\ \texttt{catch}\ (C\ V)\ e_2, s'\rangle} \tag{46}$$

$$P \vdash \langle \texttt{try Val}\ v\ \texttt{catch}\ (C\ V)\ e_2, s\rangle \xrightarrow{-} \langle \texttt{Val}\ v, s\rangle \tag{47}$$

$$\frac{hp\ s\ a\ =\ (D,fs)\quad P \vdash\ D\ \preceq^*\ C}{P \vdash \langle \texttt{try THROW}\ a\ \texttt{catch}\ (C\ V)\ e_2, s\rangle \xrightarrow{-} \langle \{V : Class\ C;\ V := addr\ a;\ e_2\}, s\rangle} \tag{48}$$

$$\frac{hp\ s\ a\ =\ (D,fs)\quad \neg\, P \vdash\ D\ \preceq^*\ C}{P \vdash \langle \texttt{try THROW}\ a\ \texttt{catch}\ (C\ V)\ e_2, s\rangle \xrightarrow{-} \langle \texttt{Throw}\ a, s\rangle} \tag{49}$$

Figure 5. Exceptional expression reduction

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).

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

The additional rules of Jinja+ are listed in Figures 7 and 8. Theres rules handle static method invocation and arrays.

Having the complete set of rules for Jinja+, we define now a run of a system.

Definition 3. 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.

$$P \vdash \langle \texttt{Cast}\ C\ (\texttt{throw}\ e), s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{50}$$

$$P \vdash \langle V := \texttt{throw}\ e, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{51}$$

$$P \vdash \langle \texttt{throw}\ e.F\{D\}, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{52}$$

$$P \vdash \langle \texttt{throw}\ e.F\{D\} := e_2, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{53}$$

$$P \vdash \langle \texttt{Val}\ v.F\{D\} := \texttt{throw}\ e, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{54}$$

$$P \vdash \langle \texttt{throw}\ e\ \ll bop \gg\ e_2, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{55}$$

$$P \vdash \langle \texttt{Val}\ v_1\ \ll bop \gg\ \texttt{throw}\ e, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{56}$$

$$P \vdash \langle \{V : T; \texttt{Throw}\ a\}_D, s\rangle \xrightarrow{D} \langle \texttt{Throw}\ a, s\rangle \tag{57}$$

$$P \vdash \langle \{V : T; V := \texttt{Val}\ v;\ \texttt{Throw}\ a\}_D, s\rangle \xrightarrow{D} \langle \texttt{Throw}\ a, s\rangle \tag{58}$$

$$P \vdash \langle \texttt{throw}\ e.M(es), s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{59}$$

$$P \vdash \langle \texttt{Val}\ v.M(map\ \texttt{Val}\ vs\ @\ (\texttt{throw}\ e\ \cdot\ es')), s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{60}$$

$$P \vdash \langle \texttt{throw}\ e;\ e_2, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{61}$$

$$P \vdash \langle \texttt{if}(\texttt{throw}\ e)\ e_1\ \texttt{else}\ e_2, s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{62}$$

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

Figure 6. Exception propagation

$$\frac{P \vdash \langle es, s\rangle\ [\xrightarrow{\ell}]\ \langle es', s'\rangle}{P \vdash \langle D.M(es), s\rangle \xrightarrow{\ell} \langle D.M(es'), s'\rangle} \tag{64}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle e[e_2], s\rangle \xrightarrow{\ell} \langle e'[e_2], s'\rangle} \tag{65}$$

$$\frac{P \vdash \langle e, s\rangle \xrightarrow{\ell} \langle e', s'\rangle}{P \vdash \langle (\texttt{Val}\ v)[e], s\rangle \xrightarrow{\ell} \langle (\texttt{Val}\ v)[e'], s'\rangle} \tag{66}$$

$$P \vdash \langle D.M(map\ \texttt{Val}\ vs\ @\ (\texttt{throw}\ e\ \cdot\ es')), s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{67}$$

$$P \vdash \langle (\texttt{throw}\ e)[e'], s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{68}$$

$$P \vdash \langle e'[\texttt{throw}\ e], s\rangle \xrightarrow{-} \langle \texttt{throw}\ e, s\rangle \tag{69}$$

Figure 7. Subexpression reduction and exception propagation rules for Jinja+.

$$\frac{P \vdash\ D\ \textit{has-static}\ M:\ Ts \to T = (pns,\ body) \qquad |vs| = |pns| \qquad |Ts| = |pns|}{P \vdash \langle D.M(map\ \texttt{Val}\ vs), s\rangle \xrightarrow{-} \langle blocks_D(pns, Ts, vs,\ body), s\rangle} \tag{70}$$

$$\frac{n \geq 0,\ \textit{new-Addr}\ h = a}{P \vdash \langle \texttt{new}\ \tau\texttt{[}intg(n)\texttt{]},\ (h,l)\rangle \to \langle addr\ a, (h(a \mapsto initArr(\tau, n)), l)\rangle} \tag{71}$$

$$P \vdash \langle null.F\{D\}, s\rangle \xrightarrow{-} \langle \texttt{THROW NullPointerException},\ s\rangle \tag{72}$$

$$\frac{n < 0}{P \vdash \langle \texttt{new}\ \tau\texttt{[}intg(n)\texttt{]},\ (h,l)\rangle \to \langle \texttt{THROW NegativeArraySizeException},\ (h,l)\rangle} \tag{73}$$

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

$$\frac{h\ a = (\tau, m, t),\ \neg(0 \leq n < m),}{P \vdash \langle (addr\ a)\texttt{[}intg\ n\texttt{]},\ (h,l)\rangle \to \langle \texttt{THROW IndexOutOfBoundsException},\ (h,l)\rangle} \tag{75}$$

$$\frac{h\ a = (\tau, m, t),}{P \vdash \langle (addr\ a)\texttt{.lenght},\ (h,l)\rangle \to \langle intg\ m,\ (h,l)\rangle} \tag{76}$$

$$\frac{h\ a = (\tau, m, t),\ 0 \leq n < m,\ isOfType(v, \tau),\ t' = arrayUpdate(t, n, v)}{P \vdash \langle (addr\ a)\texttt{[}intg\ n\texttt{]}\ \texttt{:=}\ Val\ v,\ (h,l)\rangle \to \langle Unit,\ (h(a \mapsto (\tau, m, t')), l)\rangle} \tag{77}$$

$$\frac{h\ a = (\tau, m, t),\ \neg(0 \leq n < m),}{P \vdash \langle (addr\ a)\texttt{[}intg\ n\texttt{]}\ \texttt{:=}\ Val\ v,\ (h,l)\rangle \to \langle \texttt{THROW IndexOutOfBoundsException},\ (h,l)\rangle} \tag{78}$$

$$\frac{h\ a = (\tau, m, t),\ 0 \leq n < m,\ \neg isOfType(v, \tau),}{P \vdash \langle (addr\ a)\texttt{[}intg\ n\texttt{]}\ \texttt{:=}\ Val\ v,\ (h,l)\rangle \to \langle \texttt{THROW ArrayStoreException},\ (h,l)\rangle} \tag{79}$$

Figure 8. (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 $P \vdash D\ \textit{has-static}\ M : Ts \to T = (pbs, body)$ 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 $body$.