Towards Fully Automatic Logic-Based Information Flow Analysis: An Electronic-Voting Case Study

Quoc Huy Do(✉), Eduard Kamburjan, and Nathan Wasser

Department of Computer Science, TU Darmstadt, Darmstadt, Germany
do@cs.tu-darmstadt.de, eduard.kamburjan@stud.tu-darmstadt.de,
wasser@informatik.tu-darmstadt.de

Abstract. Logic-based information flow analysis approaches generally are high precision, but lack automatic ability in the sense that they demand user interactions and user-defined specifications. To overcome this obstacle, we propose an approach that combines the strength of two available logic-based tools based on the KeY theorem prover: the KEG tool that detects information flow leaks for Java programs and a specification generation tool utilizing abstract interpretation on program logic. As a case study, we take a simplified e-voting system and show that our approach can lighten the user's workload considerably, while still keeping high precision.

Keywords: Test generation · Information flow · Invariant generation

1 Introduction

Information flow analysis has played an important role in ensuring security for software systems and has attracted many researchers for several decades. Most approaches analysing programs for secure information flow are either logic-based [4,26], which is precise but not automatic and difficult to apply for large programs, or over-approximation approaches such as type-based [1,17,24,27], dependency graph [14] or abstract interpretation [2], which are fully automatic with high performance but lack precision.

In this paper we propose a logic-based approach based on self-composition [3,10] and symbolic execution [19] that makes use of abstract interpretation [7] to obtain automation while still maintaining high precision in information flow analysis. It combines the strength of two available logic-based tools based on the KeY theorem prover: the KEG tool [12] that detects information flow leaks and a specification generation tool [28,30] utilizing abstract interpretation on program logic. The basic idea is to first analyse a target program with the specification generation tool in order to generate necessary specifications for unbounded loops

The work has been funded by the DFG priority program 1496 "'Reliably Secure Software Systems"'.

F. Piessens and L. Viganò (Eds.): POST 2016, LNCS 9635, pp. 97–115, 2016.
DOI: 10.1007/978-3-662-49635-0_6

and recursive method calls and then use the KEG tool to detect information flow leaks w.r.t. a given information flow policy. The needed loop invariants and method contracts are automatically generated by abstraction techniques, including array abstraction with symbolic pivots [30], based on abstract interpretation of partitions in an array. Loop invariants are generated without user interaction by repeated symbolic execution of the loop body and abstraction of modified variables or, in the case of arrays, the modified array elements. The invariant generation provides loop invariants which are often precise enough to be used in information leak detection.

We apply our approach in analysing two versions of a simplified e-voting system as a case study: one implementation is correct, while the other is faulty (in the sense that an information leak can happen). We show that with the correct implementation of the simplified e-voting program our approach does not report any false alarms while with the faulty implementation our approach successfully detects the leak and generates a JUnit test as witness thereof. Along with the high precision, our approach only requires users to supply a noninterference policy and preconditions for input data, but does not require any other user interactions or specifications. Our main contributions are as follows: (i) the first logic-based approach utilizing two available tools to obtain both precision and (almost) full automation in analysing information flow security, and (ii) a case study on noninterference of a simplified e-voting system showing the feasibility of our approach.

The paper is structured as follows: Sect. 2 introduces fundamental techniques used in our approach, i.e. symbolic execution and abstract interpretation. Section 3 briefly presents our logic-based leak detection approach and the implementation thereof, while the approach generating loop and method specifications is explained in Sect. 4. The combination of both tools is illustrated in Sect. 5. Section 6 demonstrates our case study and its remarks are pointed out in Sect. 7. Related work is discussed in Sect. 8 and finally Sect. 9 gives our conclusions and outlines future work.

2 Background

2.1 Symbolic Execution

Symbolic execution [19] is a powerful technique widely used in program verification, test case generation and program analysis. The main idea of symbolic execution is to run the program with symbolic input values instead of concrete ones. The central result of symbolic execution is a symbolic execution tree. Each node of the tree is annotated by its symbolic state, mapping each program location to its symbolic representation. Each path of the tree has a unique path condition that is the conjunction of all its branch conditions and represents the set of concrete executions having input values satisfying the path condition. If the program does not contain unbounded loops or recursive method calls, the symbolic execution tree is finite and covers exactly all possible concrete executions performed by the program.

In case of unbounded loops or unbounded recursive method calls a symbolic execution tree is no longer finite. One natural solution is unfolding loops or expanding invoked method calls up to a fixed depth value, generating a symbolic execution tree which is an under-approximation of the real program. Another solution is to make use of specifications as proposed in [16] to achieve a finite representation of a symbolic execution tree. This approach uses loop invariants and method contracts to describe the effect of loops and method calls. This approach gives a comprehensive view of the program's behaviour and brings scalability while still maintaining program semantics precisely. The major drawback of this approach is that specifications must be supplied in advance. In many cases this is a complex task, depending mostly on the complexity of the specific source code. Generating loop invariants as well as method specifications automatically has been an active research topic in program analysis literature.

2.2 Abstract Interpretation

Abstract interpretation [7] is a technique used in program analysis which provides a framework to lose precision within the analysis for a greater automation. When combined with symbolic execution, e.g. in [29], it allows to consider abstract symbolic values. Abstract symbolic values do not represent an unknown yet fixed concrete value, but rather in any given model the concrete value is within a set of possible concrete values – the abstract element. The abstract elements form a lattice – the abstract domain.

Given two abstract symbolic values, a_1 and a_2, the abstract domain allows to join their abstract elements, and the resulting abstract element $a_1 \sqcup a_2$ is a set which encompases at least all the possible concrete values of the two input values. This potentially loses information, as the joined element might encompase further concrete values. Information loss also occurs when abstracting a symbolic value v, i.e. finding an abstract element a such that all possible concrete values of v in a given state are within a. While information loss is inevitable, it happens in a controlled fashion and the choice of the abstract domain allows to preserve enough information for a given task, while making the set of possible values more feasable. A common choice is that the abstract domain has finite height, while the set of concrete values is infinite. This makes analysis of programs tractable and also allows fixpoint procedures, like the one presented in Sect. 4 for loop invariant generation. Abstract domains with infinite height require a widening operator in order to ensure fixpoint generation will terminate.

3 Detection of Information Flow Leaks

In this section we introduce a logic-based approach to detect (and generate exploits for) information flow leaks based on self-composition and symbolic execution that has been proposed in previous work [12] by some of the authors.

3.1 Approach

We make use of self-composition [3,10] and symbolic execution [19] to characterize and formalize information flow policies, including *noninterference* and *delimited information release* [25] as declassification. In this paper we focus on noninterference policies, the details of delimited release are explained in [12].

Given program $\mathtt{p}$ and the set V of all variables of $\mathtt{p}$, assume that V is partitioned into two subsets H and L. Program p satisfies noninterference policy $H \not\rightarrow L$ if there is no information flow from H to L. This is conventionally represented by using two program executions: $\mathtt{p}$ satisfies $H \not\rightarrow L$ iff it holds that any two executions of $\mathtt{p}$ starting in initial states that coincide on L also terminate in two states that coincide on L. Above definition can be formalized using self-composition technique proposed in [10]. A self-copied program of $\mathtt{p}$, denoted $\mathtt{p}'$, is created by copying $\mathtt{p}'$ and replacing all variables with fresh ones, such that $\mathtt{p}$ and $\mathtt{p}'$ do not share any memory. Let V', L', H' be fresh copies of V, L, H accordingly. Then $H \not\rightarrow L$ can be formalized as follows:

$$\{L \doteq L'\}\mathtt{p}(V); \mathtt{p}'(V')\{L \doteq L'\} \tag{1}$$

A major drawback of the formalization is that it requires program $\mathtt{p}$ to be analysed twice. It can be refined by making use of symbolic execution. Let SE_p and SE'_p be symbolic execution trees of $\mathtt{p}$ and $\mathtt{p}'$. It is obvious that SE_p and SE'_p are identical except that all variables $v \in V$ (considered as symbolic inputs) in all path conditions and symbolic states of SE_p are replaced by corresponding fresh copies $v' \in V'$ in SE'_p. Thus we only need to symbolically execute $\mathtt{p}$ once and represent two executions of $\mathtt{p}$ and $\mathtt{p}'$ by two symbolic execution paths of SE_p with different symbolic inputs V and V'.

For each symbolic execution path i of SE_p, we denote pc_i as its path condition. To make explicit that the symbolic final value of each program variable $v \in V$ depends on symbolic inputs and corresponding execution path, for each path i and variable v, we define function f_i^v mapping from symbolic inputs to symbolic final value of v. Let N_p be the number of symbolic execution paths of SE_p, we construct an SMT formula having the same meaning as (1):

$$\bigwedge_{0 \le i \le j < N_p} ((\bigwedge_{v \in L} v \doteq v') \wedge pc_i(V) \wedge pc_j(V') \implies \bigwedge_{l \in L} f_i^l(V) \doteq f_j^l(V')) \tag{2}$$

To detect leaks w.r.t noninterference policy $H \not\rightarrow L$, we build *insecurity formula* by negating (2) and transforming the negation into disjunctive normal form:

$$\bigvee_{l \in L} \bigvee_{0 \le i \le j < N_p} Leak(H, L, l, i, j) \tag{3}$$

where

$$Leak(H, L, l, i, j) \equiv (\bigwedge_{v \in L} v \doteq v') \wedge pc_i(V) \wedge pc_j(V') \ \wedge f_i^l(V) \not\doteq f_j^l(V') \tag{4}$$

Information flow leaks are detected by solving each formula $Leak(H, L, l, i, j)$ in (4). If it is satisfiable, there exists a forbidden information flow from some variables of H to a variable $l \in L$ and the leak can be seen by comparing two symbolic execution paths i, j. Otherwise, p is secure w.r.t the noninterference policy $H \not\rightarrow L$ if (3) is unsatisfiable.

Formula (3) can be easily extended to support detecting leaks under user-defined preconditions. Let Pre be a precondition assumed to hold at all initial states of p. To check whether p satisfies $H \not\rightarrow L$ under the assumption that Pre holds, we only need to add two conjunctions $Pre(V)$ and $Pre(V')$ into $Leak(H, L, l, i, j)$.

If p contains unbounded loops or recursive method calls, SE_p is infinite and (3) becomes unsolvable. The approach unfolding loops and expanding methods up to a fixed depth could be employed without any user interaction. Although it is useful in the sense that it can help to detect some leaks, it cannot find all possible leaks and hence cannot be used for proving secure information flow. On the other hand, the size of symbolic execution trees might be very large, thus the analysis might be very expensive. We overcome this obstacle by making use of specifications to get the finite form of SE_p as proposed in [16]. This approach represents loops and method calls as corresponding single nodes of a symbolic execution tree while keeping their semantics by using loop invariants and method contracts to contribute to relevant path conditions and to the representation of the symbolic state. For each variable v whose values can be changed during the execution of a loop or method call, its symbolic value is assigned by a fresh symbolic variable at the exit point of the loop or method call. The output value function f_i^v as well as path conditions sp_i now are represented upon $V_S = V \cup V_{fresh}$, where V_{fresh} is the set of all fresh symbolic variables created during symbolically analysing p. The approach has been implemented as a symbolic execution engine based on the verification system KeY [5], which we use as the backend for our implementation. Details and examples can be found in [12].

The precision of the information flow analysis using specifications depends mostly on the quality of the specifications. If loop invariants and method contracts are not strong enough so that they allow behaviours that are not possible in the actual program, false alarms might be raised. In the worst case when they are wrong in the sense that they exclude existing behaviours, actual leaks might not be detected. Wrong specifications can be avoided by verifying them using a program verification tool. However, refining too weak specifications is a laborious task for even an experienced user. Combining this approach with an automatic specification generation tool is a potential direction to enhance both precision and automation.

3.2 Implementation

Our approach has been implemented in a prototype tool named *KeY Exploit Generation (KEG)*[1]. KEG can automatically detect leaks in Java programs

[1] www.se.tu-darmstadt.de/research/projects/albia/download/exploit-generation-tool.

w.r.t user-specified information flow policies and generate exploits in form of JUnit tests to expose them. KEG is based on KeY [5], a state-of-the-art theorem prover for Java and makes use of its symbolic execution engine [16] which supports method and loop specifications to deal with recursive method calls and unbounded loops. KEG supports not only primitive types but also object types and arrays (to some extent). Comprehension expressions, such as sum, max and min, are also supported.

Figure 1 describes KEG's work-flow. KEG checks a Java program by analysing all specified methods w.r.t. a given information flow specification. Noninterference is a class level policy, while declassification (delimited information release) is a method level policy. To analyse a method `m`, first `m` is symbolically executed (using KeY) to achieve the symbolic execution tree. Afterwards, for each information flow policy $H \not\rightarrow L$, KEG uses the method's path conditions and the final symbolic values of the program locations modified by `m` to compose insecurity formulas $Leak(H, L, l, i, j)$. Those formulas are passed to a model finder (in our case the SMT Solver Z3 [11]) to find concrete models satisfying them. If a model has been found, it is used to configure the initial states of two runs which expose a forbidden information flow. The generated exploit then sets up two runs corresponding to two initial states and inspects the reached final values of low variables to detect a leak. KEG outputs the exploited program as an executable JUnit test.

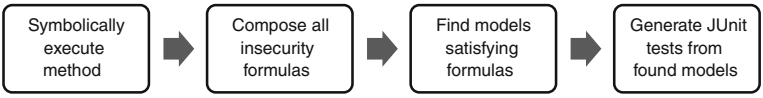


Fig. 1. Exploit generation by KEG

4 Loop Invariant Generation

4.1 KeY and Abstract Domains

The KeY tool uses symbolic execution to verify Java programs. The underlying JavaDL calculus uses *updates* [5] to encapsulate state changes of variables and models the heap memory as a special program variable. An elementary update has the form $\mathtt{x} := t$, where `x` is the program variable that is updated and t is a term which is the new value for `x`. Parallel updates are denoted $\mathcal{U} \parallel \mathcal{U}'$, where $\mathcal{U}$ and $\mathcal{U}'$ are elementary or parallel updates. Updates can be applied to terms (or formulas) with the $\{\cdot\}\cdot$ operator, resulting in new terms (or formulas). As the name implies, all elementary updates contained in a parallel update are applied simultaneously (with the rightmost update winning in case of multiple updates to the same variable). Therefore, for example, $\{\mathtt{x} := \mathtt{y} \parallel \mathtt{y} := \mathtt{x}\}(2 * \mathtt{x} + \mathtt{y})$ is equal to $(2 * \mathtt{y} + \mathtt{x})$. Updates and update applications can also be simplified: for

example, the sequential update applications $\{\mathtt{x} := \mathtt{y}\}\{\mathtt{y} := \mathtt{x}\}\phi$ can be simplified first to a parallel update application $\{\mathtt{x} := \mathtt{y} \parallel \mathtt{y} := \{\mathtt{x} := \mathtt{y}\}\mathtt{x}\}\phi$ and then the inner update application on x can be resolved, resulting in $\{\mathtt{x} := \mathtt{y} \parallel \mathtt{y} := \mathtt{y}\}\phi$. Further simplification gives $\{\mathtt{x} := \mathtt{y}\}\phi$. Updates are created during symbolic execution whenever a field or variable changes its value, e.g., this is the rule for executing variable assignments:

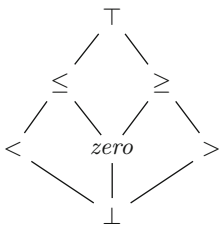
$$\textsf{assignment}\ \frac{\Gamma \Rightarrow \{\mathcal{U}\}\{\mathtt{x} := t\}[\ldots]\varphi, \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}[\mathtt{x = t;\ \ldots}]\varphi, \Delta}$$

The heap variable is updated with a special *store* function:

$$\textsf{assignment}_{array}\ \frac{\Gamma \Rightarrow \{\mathcal{U}\}\{\mathtt{heap} := \mathit{store}(\mathtt{heap}, \mathtt{a}, \mathtt{i}, t)\}[\ldots]\varphi, \Delta}{\Gamma \Rightarrow \{\mathcal{U}\}[\mathtt{a[i] = t;\ \ldots}]\varphi, \Delta}$$

Updates allow to postpone the application of state changes until the whole program has been executed and to analyze and manipulate pending state changes. The approach introduced in [6] uses the analysis of updates to incorporate abstract interpretation and loop invariant generation for local variables. We use abstract function symbols to denote abstract symbolic values, as described in Sect. 2.2.

With abstract functions it is possible to express *within* an update that, for example, an integer variable has a positive value. These abstract functions are denoted $\gamma_{\alpha,z}$, where α is the abstract element and $z \in \mathbb{Z}$ identifies the abstract function. A simple abstract domain for integers is pictured in Fig. 2.



$$\begin{array}{rll}
\top = \mathbb{Z} & & \chi_{\top}(x) = \textbf{true} \\
\leq = \{i \in \mathbb{Z} \mid i \leq 0\} & & \chi_{\leq}(x) = x \leq 0 \\
\geq = \{i \in \mathbb{Z} \mid i \geq 0\} & & \chi_{\geq}(x) = x \geq 0 \\
< = \{i \in \mathbb{Z} \mid i < 0\} & & \chi_{>}(x) = x > 0 \\
> = \{i \in \mathbb{Z} \mid i > 0\} & & \chi_{<}(x) = x < 0 \\
\mathit{zero} = \{0\} & & \chi_{\mathit{zero}}(x) = x \doteq 0 \\
\emptyset = \{\} & & \chi_{\bot}(x) = \textbf{false}
\end{array}$$

Fig. 2. Abstract domain for integers

The lattice structure allows joining updates into abstract updates that describe a set of possible value changes. E.g., the update $\mathtt{x} := \gamma_{>,1} \parallel \mathtt{y} := \gamma_{>,2}$ sets x to some positive value and y to another, possibly different, positive value. The additional information of the $\gamma_{\alpha,z}$ symbols can be obtained by adding the description of α to the premiss of the sequent. The description of each α is contained in the matching characteristic function χ_{α}.

4.2 Generation and Implementation

In [6] abstract interpretation is used to generate loop invariants for local variables. To do this, the loop is symbolically executed once and the resulting symbolic program states are joined with the initial (symbolic) program state: For each variable, the value in the update is abstracted, i.e. the smallest abstract element that contains all possible values is determined. Then all abstract elements are joined. This is repeated until a fixpoint is found, i.e. another iteration does not produce a weaker update.

Example 1. Consider the sequent

$$\Rightarrow \{\texttt{i} := 1\}[\textbf{while}\texttt{(i > 0) i = i-1;}](\texttt{i} = 0)$$

The initial program state is expressed in the update $\texttt{i} := 0$ and symbolic execution of one iteration leads to the sequent

$$\Rightarrow \{\texttt{i} := 0\}[\textbf{while}\texttt{(i > 0) i = i-1;}](\texttt{i} = 0)$$

Both 1 and 0 are contained in $\geq$ thus the updates are joined to $\texttt{i} := \gamma_{\geq,0}$, which is used as the pre-state in the next iteration. Another iteration produces the update $\texttt{i} := \gamma_{\geq,0} - 1$ under the premiss that $\gamma_{\geq,0} > 0$ and leads to no weaker update.

Fields of integer type can be handled analogously and simple domains for boolean and object variables/fields can be used.

In [29] this approach was extended to abstraction of arrays: Arrays are regarded as split into two parts: a (potentially) modified and an unmodified part. If the sequence of array accesses is monotonously increasing (or decreasing), then each iteration moves the splitting point further and allows to abstract all the array elements in between. With this method it is possible to generate invariants of the form

$$\forall i.\ (\mathit{initial} \leq i \wedge i < \mathit{id}) \rightarrow \chi_a(\texttt{arr}[i])$$

where *id* is the index term for the array access and *initial* is the value of *id* before the loop.

If in the loop the sequence of array access has the form $a + x * b$ in the xth iteration of the loop for some $a, b \in \mathbb{Z}$, then a more precise invariant is possible that only makes a statement about the elements actually accessed. If the sequence is not monotonous, a very weak invariant of the form

$$\forall i.\ (0 \leq i \wedge i < \texttt{arr.length}) \rightarrow (\chi_a(\texttt{arr}[i]) \vee \texttt{arr}[i] = \texttt{arr}_{\mathit{old}}[i])$$

can be generated, where $\texttt{arr}_{\mathit{old}}$ denotes the array in the state before entering the loop for the first time.

Additional forms of invariants can be extracted from the unrolled loops in the invariant generation for arrays and variables. These include simple abstraction over arbitrary terms, that can be used, e.g., to establish an order between

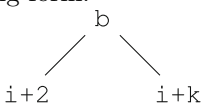
variables. Another extension is *sum invariants* that can be generated if a variable is modified only by summing another value inside the loop. In this case after symbolic execution of the fixed point iteration all open branches contain an update of the form $\mathtt{x} := \gamma_{\alpha,z} + t$ for the variable $\mathtt{x}$ in question, where $\gamma_{\alpha,z}$ is the value of $\mathtt{x}$ before execution of the loop body. (Also accepted are updates $\mathtt{x} := \gamma_{\alpha,z}$ and $\mathtt{x} := \gamma_{\alpha,z} - t$.) As an additional requirement all variables in t must have the aforementioned form $a + x * b$ in the xth iteration for some $a, b \in \mathbb{Z}$.

As it is possible that different terms are added to the variable, the execution tree is condensed into a tree that only contains the splitting nodes and the post-states. The splitting conditions in the inner nodes are used as conditions in the ternary conditional expression operator when constructing the sum formula.

Example 2. Consider the sequent

$$\Rightarrow \{\mathtt{i} := 1 \parallel \mathtt{j} := 2 \parallel \mathtt{k} := 5\}[\textbf{while}(k > 0)\{ \textbf{if}(\mathtt{b})\ \mathtt{i = i + j;}\ \textbf{else}\ \mathtt{i = i + k;}\ \mathtt{k{-}{-};}\}](\mathtt{k} = 0)$$

This generates the invariant update $\mathtt{i} := \gamma_{>,z} \parallel \mathtt{j} := 2 \parallel \mathtt{k} := \gamma_{\geq,z}$ and the condensed tree has the following form:



As $\mathtt{k}$ has the form $5 - x$ in the xth iteration, it produces the invariant

$$\mathtt{i} = 1 + \sum_{\mathtt{n=0}}^{it} \mathtt{b\ ?\ 5{-}n\ :\ 2}$$

The invariant generation uses the generated invariant directly in order to potentially reach other loops or the same loop in a different program state and generate further loop invariants, but also outputs JML [22]. When the invariant is used in a proof, by application of a loop invariant rule, it is ensured that the invariant is correct, i.e. it holds before the first execution and holds after every iteration if it held before.

As the invariant consists of subformulas with fixed form, each is translated separately:

- Updates of the form $\mathtt{x} := \gamma_{\alpha,z}$ are translated into $\chi_\alpha(\mathtt{x})$. The χ-functions can be rewritten to JML formulas, e.g., $\chi_>(\mathtt{x})$ would become $\mathtt{x} > 0$. If several variables share the same γ-constant, the corresponding equalities are added.
- Array invariants use the same rewriting of χ-functions.
- Sum invariants are translated into JML with the \sum operator. E.g., the sum invariant in Example 2 is translated to

$$\mathtt{i} = 1 + \backslash\mathtt{sum}\ \textbf{int}\ n;\ n >= 0\ \&\&\ n < \mathtt{iter};\ \mathtt{b\ ?\ 5{-}n\ :\ 2}$$

If a subformula is equal to **true**, e.g. $\chi_\top(\mathtt{i})$, it is omitted in the JML output.

5 Fully Automatic Approach

The logic-based information flow analysis approach proposed in Sect. 3 requires that specifications necessary for information flow analysis, i.e. loop invariants and method contracts, must be supplied by the user. This is usually a tough task and requires a considerable effort. In this section we demonstrate an approach that reduces the workload of the user towards obtaining a fully automatic analysis of information flow for Java programs. The fundamental idea is that we leave the task of generating loop invariants and method contracts to the tool proposed in Sect. 4 and use these generated specifications in the information flow analysis by KEG.

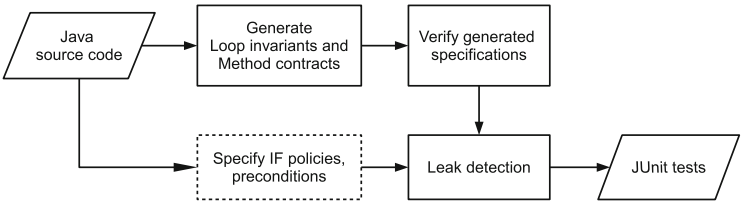


Fig. 3. Fully automatic leak detection for java programs

Figure 3 shows the combination of two tools to automatically detect information leaks in a Java program. The solid border rectangle boxes represent automatic actions performed by our tools, while the dashed border one is for manual action done by the user. If the Java program contains unbounded loops and/or recursive method calls, the specification generator is activated to generate corresponding specifications and insert them into the original source code. Generated specifications are also verified by a verification tool, here we use the theorem prover KeY. Finally, the specified program is automatically analysed w.r.t. user-defined information flow policies and other specifications (usually preconditions) using KEG to create JUnit tests helping to demonstrate discovered leaks as well as serving for regression tests.

6 E-Voting Case Study

In this section we present our case study on verifying the privacy property of an e-voting system by proving the noninterference property of a simplified, ideal Java counterpart[2].

[2] www.se.tu-darmstadt.de/research/projects/albia/download/e-voting-case-study/.

6.1 From Privacy to Noninterference

Our case study is a modified, extended version of the e-voting case study introduced in [20,21]. In order to prove the cryptographic privacy of the votes of honest voters, the authors constructed a cryptographic privacy game formulated in Java. In that game, the environment (the adversary) can provide two vectors $\overline{c_0}$ and $\overline{c_1}$ of choices of voters such that the two vectors yield the same result according to the counting function, otherwise the game is stopped immediately. Afterwards, the voters vote according to $\overline{c_b}$, where b is a secret bit. The adversary tries to distinguish whether the voters voted according to $\overline{c_0}$ or to $\overline{c_1}$. If they succeed, the cryptographic privacy property is broken. By defining this game, instead of proving the cryptographic privacy property of the complex e-voting system, the authors of [21] prove the noninterference property of its ideal simplified counterpart, which states that there is no information flow from secret bit b to the public result on the bulletin board. It states that if the voting machine computes the result correctly, then this result is independent of whether the voters voted according to $\overline{c_0}$ or $\overline{c_1}$.

We re-implement the simplified version of the e-voting system in [20] by a slightly more complicated version in which the system can handle an arbitrary number of candidates rather than only two. Figure 4 depicts the core of our case study program that includes two classes: `Result` wraps the result of the election and `SimplifiedEvoting` reproduces the privacy game mentioned in [21]. Class `Result` has one public integer array field `bulletin`, where `bulletin[i]` stores the number of votes for candidate i. Class `SimplifiedEVoting` has the following fields: a private logic variable `secret` as the secret bit, an integer variable `n` representing the number of candidates indexed by n consecutive integer number from 0 to $n-1$; two integer arrays `votesX`, `votesY` as two vectors of votes supplied by the adversary, where each array's element i is an integer number j (ideally $0 \leq j \leq n-1$) which mean that voter i votes for candidate j; and finally the public variable `Result` that can be observed by the adversary. Method `privacyGame` of class `SimplifiedEvoting` mimics the process that the result is computed using one of two vectors of votes based on the value of the secret bit. Method `compute` of class `SimplifiedEvoting` computes the result of the election using the corresponding vector of votes passed as its parameter. Line 7 is the noninterference policy claiming that there is no information flow from `secret` to `result`. To deal with this object-sensitive noninterference policy, we implement the approach introduced in [4]. We experiment using our approach on two versions of `compute`: one is a correct implementation, while the other is faulty.

The precondition of method `privacyGame` is depicted in Fig. 5, enforcing that two vectors of votes (`votesX` and `votesY`) have the same size and produce the same result before `privacyGame` is executed. It also makes sure that the number of candidates is greater than 1 and every single vote belongs to one of those candidates.

```
1  public class SimplifiedEVoting {
2    private boolean secret;
3    public Result result;
4    int n; //number of candidates
5    int[] votesX, votesY;
6
7    /*! result | secret ; !*/
8
9    private Result compute(int[] votes){
10     /* implementation of compute  */
11   }
12
13   /*@requires ... @*/
14   public void privacyGame(){
15     if(secret)
16       result = compute(votesX);
17     else
18       result = compute(votesY);
19   }
20 }
21
22 public class Result {
23   public int[] bulletin;//result of votes
24   public Result(int n) {
25     if(n>0)
26       this.bulletin = new int[n];
27     else
28       this.bulletin = null;
29   }
30 }
```

Fig. 4. Simplified e-voting program

```
1 /*@requires votesX!=null && votesY != null
2   && (votesX.length == votesY.length) && (votesX.length>0) && n>=2
3   && (\forall int j; j>=0 && j<votesX.length;
4         votesX[j]>=0 && votesX[j]<n && votesY[j]>=0 && votesY[j]<n)
5   && (\forall int i; 0 <= i && i < n;
6   (\sum int j; 0 <= j && j < votesX.length; (votesX[j]==i ? 1 : 0))
        ==
7   (\sum int j; 0 <= j && j < votesY.length; (votesY[j]==i ? 1 : 0)))
        ;
8  @ diverges true;
9 @*/
```

Fig. 5. Precondition as JML specification of method privacyGame

6.2 Leak Detection for Correct Implementation

We first show the result of our approach for the correct implementation of method `compute` as shown in Fig. 6. To check the security of the method `privacyGame`, it is first symbolically executed by the KeY tool. The input file is shown in Fig. 7.

This first step symbolically executes the loop 7 times in total, opens 105 side proofs and needs 148 s on a i5-3210M CPU with 6 GB RAM. As our system is not optimized for speed, we suppose that it is possible to generate the invariants in significantly less time. The output of the symbolic execution is, besides the proof tree, a file named SimplifiedEVoting.java.mod.0, which contains the Java file with the annotations. In Fig. 8 the result of the loop invariant generation for

```
private Result compute(int[] votes){
  Result rs = new Result(n);
  for(int i = 0; i< votes.length; i++){
    if(votes[i]>=0 && votes[i]<n)
      rs.bulletin[votes[i]] = rs.bulletin[votes[i]] + 1;
  }
  return rs;
}
```

Fig. 6. Correct implementation of method `compute`

```
    \javaSource ".";

    \programVariables{
        SimplifiedEVoting2 vt;
        int[] v;
        Result2 r;
    }

    \problem{
        vt != null &
        vt.<created> = TRUE &
        vt.<inv> &
        vt.n >0 &
        v != null &
        v.<created> = TRUE &
        wellFormed(heap) &
        (\forall int i; (i >=0 & i <v.length -> (v[i] >= 0 & v[i] <
            vt.n))) &
        r = null
        ->
        \[{
                r = vt.compute(v);
        }\](r != null)

    }
```

Fig. 7. Inputfile input.key

the loop in method `compute` is depicted. The invariant is generated by calling the method `compute`, not by calling the method `privacyGame`, because the loop invariant generation is *local*, in the sense that it produces invariants valid under a given precondition. Calling `privacyGame` would produce two invariants, one for each branch, which must be combined using the splitting condition distinguishing them. This may lose precision because the splitting condition may be not fully known, thus the generating call should be to the method containing the loop.

In the next step, the file SimplifiedEVoting.java.mod.0 is renamed to SimplifiedEVoting.java and used as input for the KEG tool. KEG finished checking the program w.r.t noninterference policy in 41 s on the same system without finding any information flow leak.

6.3 Leak Detection for Faulty Implementation

Now we change the implemenation of method `compute` slightly, such that it ignores the first element in the vector of votes when calculating the result.

```
1     //@ghost int iter = 0;// AUTO_GENERATED BY KeY
2     /*@ // AUTO_GENERATED BY KeY
3        loop_invariant
4         rs.bulletin == rs.bulletin
5     && i >= 0
6     && i == (\sum int q; 0 <= q & q <iter;1 + 0)
7     &&( \forall int j_27;
8         (   0 <= j_27 & j_27 < rs.bulletin.length
9          ==>   (\sum int q_1; 0 <= q_1 & q_1 <iter; (votes[q_1] ==
                j_27)
10                                  ? (1 + 0)
11                                  : (0))
12            == rs.bulletin[j_27]))
13    && (iter >= 0 & iter * 1 == i)
14    && i - votes.length <= 0
15    &&( \forall int j_28;
16        (j_28 < i & 0 <= j_28 ==> votes[j_28] >= 0)
17    );
18       assignable
19       rs.bulletin[*],i;
20    @*/
21
22        for(int i = 0; i< votes.length; i++){
23    //@set iter = iter + 1;// AUTO_GENERATED BY KeY
24                if(votes[i]>=0 && votes[i]<n)
25                   rs.bulletin[votes[i]] = rs.bulletin[votes[i]] +
                        1;
26         }
27           return rs;
28       }
```

Fig. 8. Annotated SimplifiedEVoting.java.mod.0

It is obviously an incorrect implementation, in that two vector of votes `votesX`, `votesY` can produce two different results even if the precondition of method `privacyGame` holds. The faulty implementation is given in Fig. 9.

```
1   private Result compute(int[] votes){
2     Result rs = new Result(n);
3     for(int i = 1; i< votes.length; i++){ //omit votes[0]
4       if(votes[i]>=0 && votes[i]<n)
5         rs.bulletin[votes[i]] = rs.bulletin[votes[i]] + 1;
6     }
7     return rs;
8   }
```

Fig. 9. Faulty implementation of method `compute`

For this method, the loop invariant generation opens 86 side proofs, executes the loop 7 times in total and needs 161 s on a i5-3210M CPU with 6 GB RAM.

The KEG tool finishes checking method `privacyGame` calling the faulty implementation of `compute` in 145 s and finds a leak. It reports that there is an implicit information flow leak caused by two different symbolic execution paths branched by the value of `secret`. Using precondition of method `privacyGame` as in Fig. 5, KEG generates input values for `votesX` and `votesY` in order to demonstrate the leak as follows:

array	element at index 0 1 2 3 4 5 6 7 8
votesX	1 2 2 1 1 0 0 1 0
votesY	2 1 1 1 1 0 0 0 2

It is easy to see that the generated values of `votesX` and `votesY` bring the same election result by using the correct version of `compute`, however the results computed by the faulty method `compute` differ. This helps the attacker infer the value of bit `secret` and break the privacy property of the e-voting system.

7 Discussion

We chose the simplified e-voting system as case study for our approach for the following reasons: (i) its noninterference property has been verified using a hybrid approach [21] that is not automatic and requires the program to be modified; (ii) it is a sequential Java program having complex features of real-life object oriented programs such as reference types, arrays and object creation; and (iii) the program requires complex specifications containing comprehension sum that challenge both our specification generation tool and the KEG tool.

Comprehension expressions like sum, max and min are usually not natively supported by SMT Solver. KEG uses the SMT Solver Z3 to solve insecurity formulas. While Z3 is very powerful, it does not natively support comprehension expressions. KEG treats sum in a similar way to the approach proposed in [23], where each sum is translated into a self-contained function characterized by its axioms. The original implementation for the translation of sum (and other comprehension expressions such as max and min) binds each expression to a corresponding function that has two parameters describing the interval. For example, consider the following sum expression in JML syntax:

```
(\sum int i; 0 <= i && i < votes.length; votes[i])
```

This can be translated into a function call `sum_0(0, votes.length-1)`, where `sum_0` is characterized by the following axioms:

$$\begin{aligned}
&\forall \mathtt{x}, \mathtt{y} \in \{0, 1, .., \mathtt{votes.length} - 1\} : \\
&\quad \mathtt{x} > \mathtt{y} \Rightarrow \mathtt{sum_0}(\mathtt{x}, \mathtt{y}) = 0 \wedge \\
&\quad \mathtt{x} = \mathtt{y} \Rightarrow \mathtt{sum_0}(\mathtt{x}, \mathtt{y}) = \mathtt{votes}[\mathtt{x}] \wedge \\
&\quad \mathtt{x} < \mathtt{y} \Rightarrow \mathtt{sum_0}(\mathtt{x}, \mathtt{y}) = \mathtt{votes}[\mathtt{x}] + \mathtt{sum_0}(\mathtt{x} + 1, \mathtt{y})
\end{aligned}$$

This translation approach is simple but versatile and can be used for all types of comprehension expressions. The drawback of this approach is that it does not support quantification, i.e. if sum is nested in a universal expression (as shown at lines 3 - 7 in Fig. 5). To solve this problem, we tailor a new translation approach for sum if it is quantified. We extend the generated sum functions with a parameter representing the quantified variable. For example, following quantified clause in the precondition shown in Fig. 5:

```
(\forall int i; 0 <= i && i < n;
(\sum int j; 0<=j && j<votesX.length; (votesX[j]==i?1:0))
   ==
(\sum int j; 0<=j && j<votesY.length; (votesY[j]==i?1:0)))
```

can be translated into following expression:

$$\forall\, \mathtt{i} \in \{0, 1, .., \mathtt{n}-1\}:$$
$$\mathtt{sum\_1}(0, \mathtt{votesX.length}-1, \mathtt{i}) = \mathtt{sum\_2}(0, \mathtt{votesY.length}-1, \mathtt{i})$$

The corresponding axioms chracterising `sum_1` and `sum_2` are also added into the insecurity formula. Although this approach allows quantifying over sum expressions (also other comprehensions), it is not suitable for all instances of sum and brings considerable extra workload for the SMT Solver. We do believe that there is no one-size-fits-all method translating comprehension expressions to SMT first order formulas that exists and it is necessary to optimize the translation w.r.t. each specific case.

The ability to generate invariants containing comprehension expressions (in this case sum) was crucial in this case study in order to generate a strong enough functional invariant so as to be able to prove noninterference with the KEG tool. Comprehension expressions also allow the ability to be much more precise about the value of a variable or array index term, rather than using abstraction, which is often only an over-approximation.

In general, sums and arrays interact with each other quite nicely, in that (i) programs often sum values based on the elements in an array (besides the example in this case study another simple example would be calculating the sum of all elements), but also (ii) array index terms are often sums (sometimes also expressible as an affine term, but for example a binary tree expressed in array form gets from index i to its left child node by adding $i + 1$ to the index).

Analysis of whether a comprehension expression can be used to express an invariant is quite simple in our tool, as we can see the program updates and branch conditions of all sequents resulting from symbolic execution of the loop body, thus infering sums, etc. from the actual symbolic values, rather than trying to syntactically analyse the program code.

8 Related Work

Our e-voting case study is motivated and based on the one used in [20,21]. In that paper, the authors propose a hybrid approach combining the strengths of an automatic tool (Joana [14]) and a deductive verification tool (KeY [5]) for proving noninterference property of a simple e-voting system. This approach requires programs to be extended so that the automatic tool for proving noninterference can be applied without returning any false positives. The deductive verification tool is used to prove functional properties of the e-voting program and its extended version. While enhancing the precision of the automatic tool Joana, the approach still needs a lot of user interaction in establishing and proving functional properties as program invariants.

There has been a lot of research into secure information flow. Some logic-based approaches such as [4,26] are fully precise but not fully automatic in the sense that they require from the user not only specifications for unbounded loops and recursive method calls but also non-trivial interactions with the theorem prover. On the other hand, approaches based on type systems [1,17,24,27] or those based on dependency graphs [14] are fully automatic and able to check real-life programs due to their high performance. However, these approaches share common drawbacks of over-approximation on actual information flow that lead to lack of precision and resulting false positives in many cases.

Several tools for loop invariant generation have been proposed and using abstract interpretation is among the first approaches [9]. Such tools were developed for other theorem provers, e.g. ESC/Java2 [13,18], but concentrate on checking bounds when dealing with arrays. Other approaches which are more precise concerning arrays either rely on syntactic analysis and restrictions [15] or on additional information provided together with the abstract domain [8].

9 Conclusion

We proposed a novel logic-based approach towards fully automatic information flow analysis by combining the strength of two logic-based tools. We applied it for a simplified version of an e-voting system as case study to check noninterference policy that is the counterpart of cryptographic privacy property. By the case study result, we showed that our approach is not only precise (it can detect the potential leak of an insecure program while not raising false positives for a secure program) but also automatic (it only requires user to supply expressive information flow policy and precondition describing the constraint of the initial program state). Although the case study revolves around a relatively small program, it is not a simple program and it is sufficient for exposing the strengths as well as limitations of our tools, which shows that our approach is very promising to be used for real-life programs.

For future work, we aim to extend our tools and their combination towards analysing real-life programs, which are usually large and complex. A potential solution is to use method contracts instead of simply expanding method calls, which brings compositionability, scalability and analysis re-usability. Both tools we used are adequate for this direction, in that the specification generation tool can already generate method contracts for recursive methods [28], while the KEG tool can use method contracts for leak detection. However, they need to be improved in performance as well as expanding the set of language features they support. Optimizing the specification generation towards supporting better information flow analysis is another promising direction.

Acknowledgements. We would like to thank Richard Bubel for fruitful discussions and comments.

References

1. Avvenuti, M., Bernardeschi, C., Francesco, N.D., Masci, P.: JCSI: a tool for checking secure information flow in java card applications. J. Syst. Softw. **85**(11), 2479–2493 (2012)
2. Banerjee, A., Giacobazzi, R., Mastroeni, I.: What you lose is what you leak: information leakage in declassification policies. Electron. Notes Theor. Comput. Sci. **173**, 47–66 (2007)
3. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: Proceedings of the 17th IEEE Workshop on Computer Security Foundations, CSFW 2004, pp. 100–114. IEEE CS(2004)
4. Beckert, B., Bruns, D., Klebanov, V., Scheben, C., Schmitt, P.H., Ulbrich, M.: Information flow in object-oriented software. In: Gupta, G., Peña, R. (eds.) LOPSTR 2013, LNCS 8901. LNCS, vol. 8901, pp. 19–37. Springer, Heidelberg (2014)
5. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. Lecture Notes in Computer Science, vol. 4334. Springer, Heidelberg (2007)
6. Bubel, R., Hähnle, R., Weiß, B.: Abstract interpretation of symbolic execution with explicit state updates. In: de Boer, F.S., Bonsangue, M.M., Madelaine, E. (eds.) FMCO 2008. LNCS, vol. 5751, pp. 247–277. Springer, Heidelberg (2009)
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4th Symposium on Principles of Programming Languages (POPL), pp. 238–252. ACM (1977)
8. Cousot, P., Cousot, R., Logozzo, F.: A parametric segmentation functor for fully automatic and scalable array content analysis. SIGPLAN Not. **46**(1), 105–118 (2011)
9. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL 1978, pp. 84–96. ACM (1978)
10. Darvas, Á., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: Gorrieri, R. (ed.) Workshop on Issues in the Theory of Security. IFIP WG 1.7, SIGPLAN and GI FoMSESS. ACM (2003)
11. de Moura, L., Bjørner, N.S.: Z3: an efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
12. Do, Q., Bubel, R., Hähnle, R.: Exploit generation for information flow leaks in object oriented programs. In: Federrath, H., Gollmann, D. (eds.) ICT SystemsSecurity and Privacy Protection. IFIP Advances in Information and Communication Technology, vol. 455, pp. 401–415. Springer, Heidelberg (2015)
13. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. SIGPLAN Not. **37**(1), 191–202 (2002)
14. Graf, J., Hecker, M., Mohr, M.: Using JOANA for information flow control in java programs - a practical guide. In: Proceedings of the 6th Working Conference on Programming Languages. LNI, vol. 215, pp. 123–138. Springer, February 2013
15. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. SIGPLAN Not. **43**(6), 339–348 (2008)
16. Hentschel, M., Hähnle, R., Bubel, R.: Visualizing unbounded symbolic execution. In: Seidl, M., Tillmann, N. (eds.) TAP 2014. LNCS, vol. 8570, pp. 82–98. Springer, Heidelberg (2014)

17. Hunt, S., Sands, D.: On flow-sensitive security types. In: ACM SIGPLAN Notices, vol. 41, pp. 79–90. ACM (2006)
18. Janota, M.: Assertion-based loop invariant generation. In: Proceedings of the 1st International Workshop on Invariant Generation (WING 07), Wing 2004 (2007)
19. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
20. Küsters, R., Truderung, T., Beckert, B., Bruns, D., Graf, J., Scheben, C.: A hybrid approach for proving noninterference and applications to the cryptographic verification of java programs. In: Grande Region Security and Reliability Day 2013, Extended Abstract (2013)
21. Küsters, R., Truderung, T., Beckert, B., Bruns, D., Kirsten, M., Mohr, M.: A hybrid approach for proving noninterference of java programs. In: Fournet, C., Hicks, M. (eds.) 28th IEEE Computer Security Foundations Symposium (2015)
22. Leavens, G.T., Baker, A.L., Ruby, C.: JML: a java modeling language. In: Formal Underpinnings of Java Workshop (at OOPSLA 1998), pp. 404–420 (1998)
23. Leino, K.R.M., Monahan, R.: Reasoning about comprehensions with first-order SMT solvers. In: Proceedings of the 2009 ACM Symposium on Applied Computing, SAC 2009, pp. 615–622. ACM, New York (2009)
24. Myers, A.C.: JFlow: practical mostly-static information flow control. In: Proceedings of 26th ACM Symposium on Principles of Programming Languages, pp. 228–241 (1999)
25. Sabelfeld, A., Myers, A.C.: A model for delimited information release. In: Futatsugi, K., Mizoguchi, F., Yonezaki, N. (eds.) Software Security - Theories and Systems. Lecture Notes in Computer Science, vol. 3233, pp. 174–191. Springer, Heidelberg (2004)
26. Scheben, C., Schmitt, P.H.: Verification of information flow properties of JAVA programs without approximations. In: Beckert, B., Damiani, F., Gurov, D. (eds.) FoVeOOS 2011. LNCS, vol. 7421, pp. 232–249. Springer, Heidelberg (2012)
27. Volpano, D., Irvine, C., Smith, G.: A sound type system for secure flow analysis. J. Comput. Secur. **4**(2), 167–187 (1996)
28. Wasser, N.: Generating specifications for recursive methods by abstracting program states. In: Li, X., Liu, Z., Yi, W. (eds.) Dependable Software Engineering: Theories, Tools, and Applications. Lecture Notes in Computer Science, vol. 9409, pp. 243–257. Springer, Heidelberg (2015)
29. Wasser, N., Bubel, R.: A theorem prover backed approach to array abstraction. Technical. report, Department of Computer Science, Technische Universität Darmstadt, Germany , presented at the Vienna Summer of Logic 2014 5th International Workshop on Invariant Generation (2014)
30. Wasser, N., Bubel, R., Hähnle, R.: Array abstraction with symbolic pivots. Technical report, Department of Computer Science, Technische Universität Darmstadt, Germany, August 2015