Exploit Generation for Information Flow Leaks in Object-Oriented Programs

Quoc Huy Do(✉), Richard Bubel, and Reiner Hähnle

Department of Computer Science, TU Darmstadt, Darmstadt, Germany
{do,bubel,haehnle}@cs.tu-darmstadt.de

Abstract. We present a method to generate automatically exploits for information flow leaks in object-oriented programs. Our approach combines self-composition and symbolic execution to compose an *insecurity formula* for a given information flow policy and a specification of the security level of the program locations. The insecurity formula gives then rise to a model which is used to generate input data for the exploit.

A prototype tool called KEG implementing the described approach for Java programs has been developed, which generates exploits as executable JUnit tests.

Keywords: Test generation · Symbolic execution · Information flow

1 Introduction

Analyzing programs to ensure that they do not leak secrets is necessary to improve confidence in the ability of a system to not put the security and privacy of its users at stake.

Information flow analysis is concerned with one aspect of this task, namely, to ensure that an outside agent with well-defined properties cannot learn secrets by observing (and initiating) several runs of a program. The nature of the secrets to be protected is specified by an information flow policy. The strongest one is noninterference, which does not allow the attacker to learn *any* kind of information about the secret. This is often too strong, e.g., an authentication program leaks the information whether an entered password is correct, hence, other policies like declassification (see [22] for a survey) exist that allow to specify what kind of information may be released.

Several approaches to analyze programs for secure information flow relative to a given information flow policy exist. Many of these are either type-based [4,15,18,21,26] or logic-based [6,11,23]. In this paper we use a logic-based approach with self-composition (first introduced in [10]; the name self-composition was coined in [6]), but our focus is not to verify that a program has secure information flow; instead we approach the problem from a bug finding point of view.

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

© IFIP International Federation for Information Processing 2015
H. Federrath and D. Gollmann (Eds.): SEC 2015, IFIP AICT 455, pp. 401–415, 2015.
DOI: 10.1007/978-3-319-18467-8_27

For a given program we try to automatically generate exploits that demonstrate unintended information flows. Exploits are small programs that run the program of interest multiple times and report whether they could observe a leak. The generated exploits are well-structured and support the developer in identifying the origin of the leak and in understanding its nature.

To generate exploits we build on work from test generation [12,16] with symbolic execution. Our implementation outputs the found exploits as JUnit tests such that the test fails if the program is insecure. The exploits can thus easily be added to a regression test suite.

The paper is structured as follows: Sect. 2 introduces basic notions and techniques. Sect. 3 explains the logic formalization of insecurity for noninterference and delimited information release. In Sect. 4 we discuss the analysis of programs containing loops and method invocations. Sect. 5 presents our tool KEG and demonstrates the viability with case studies. We compare our work with others in Sect. 6 and conclude with Sect. 7.

2 Background

2.1 Information Flow Policies

To analyze that a program does not leak confidential information, we need to define the security level of the program locations (program variables and fields) as well as an *information flow policy* which defines whether and what kind of information may flow between program locations of a different security level.

In this subsection we recapture the definitions of two well-known information flow policies which are supported by our approach.

Noninterference. Noninterference [9,26] is the strongest possible information flow policy. It prohibits *any* information flow from program locations containing confidential information (high variables) to publicly observable program locations (low variables); the opposite direction is allowed. As we consider only deterministic programs, noninterference can be easily formalized by comparing two program runs:

A program has secure information flow with respect to noninterference, if any two executions of the program starting in initial states with identical values of the low variables, also end in final states which coincide on the values of the low variables.

Let $\mathtt{p}$ denote a program and *Var* the set of all program variables of $\mathtt{p}$.

Definition 1 (Program State). *A* program state σ *maps each program variable* $v \in Var$ *of type* T *to a value of its concrete domain* D^T*, i.e.,*

$$\sigma : Var \to D$$

with $\sigma(v : T) \in D^T$ *and* D *being the union of all concrete domains. The set of all states for a given program* $\mathtt{p}$ *is denoted as* $States_{\mathtt{p}}$*.*

We define coincidence of program states w.r.t. a set of program variables:

Definition 2 (State Coincidence). *Given a set of program variables V and two states $\sigma^1, \sigma^2 \in States_{\mathtt{p}}$. We write $\sigma^1 \simeq_V \sigma^2$ iff. σ^1 and σ^2 coincide on V, i.e., $\sigma^1(v) = \sigma^2(v)$ for all $v \in V$.*

A *concrete execution trace* τ of a program $\mathtt{p}$ is a possibly infinite sequence of program states $\tau = \sigma_0\sigma_1 \ldots$ produced by starting $\mathtt{p}$ in state σ_0. In this paper, we are only concerned with terminating programs, and consequently, all of our execution traces are finite. Thus, we represent a concrete execution X of a program $\mathtt{p}$ as tuple $\langle \sigma^X, \sigma^X_{out} \rangle$, where $\sigma^X \in States_{\mathtt{p}}$ is the initial program state and $\sigma^X_{out} \in States_{\mathtt{p}}$ is the final program state. The set of all possible concrete executions of $\mathtt{p}$ is denoted as Exc_p.

We can now define the noninterference property as follows:

Definition 3 (Noninterference). *Given a* noninterference policy $NI = (L, H)$ *where $L \dot\cup H = Var$ s.t. L contains the low variables and H the high variables.*

A program $\mathtt{p}$ has secure information flow with respect to NI iff. for all concrete executions $X, Y \in Exc_p$ it holds that if $\sigma^X \simeq_L \sigma^Y$ then $\sigma^X_{out} \simeq_L \sigma^Y_{out}$.

Declassification. For many practical cases noninterference is too strict. E.g., a login program leaks usually the information whether an entered password is correct; or a database may be queried for aggregated information like the average salary of the employees, but not for the income of an individual employee.

Declassification is a class of information flow policies which allows to express that some confidential information may be leaked under specific conditions. The paper [22] provides an extensive survey of declassification approaches.

In this paper we focus on *delimited information release* as introduced in [21]. Delimited information release is a declassification policy which allows to specify what kind of information may be released. To this end so called *escape hatch* expressions are specified in addition to the security level of the program locations. For instance, the escape hatch $\frac{\sum_{e \in Employees} salary(e)}{|Employees|}$ can be used to declassify the average of the income of all employees. The formal definition of delimited information release is similar to Def. 3:

Definition 4 (Delimited Information Release). *Given a* delimited information release policy $Decl = (L, H, E)$ *with L, H as before and E denoting a set of escape hatch expressions.*

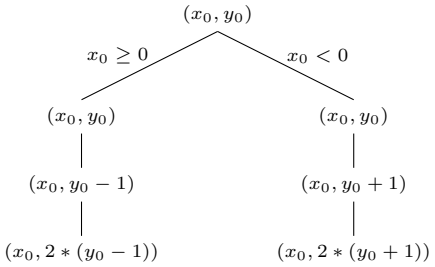
A program $\mathtt{p}$ has secure information flow with respect to Decl iff. for all concrete executions $X, Y \in Exc_p$ it holds that if $\sigma^X \simeq_L \sigma^Y$ and for all $e \in E : [\![e]\!]_{\sigma^X} = [\![e]\!]_{\sigma^Y}$ then $\sigma^X_{out} \simeq_L \sigma^Y_{out}$. The expression $[\![e]\!]_\sigma$ denotes the semantic evaluation of e in state σ.

2.2 Logic-Based Information Flow Analysis

Symbolic Execution. Symbolic Execution [16] is a versatile technique used for various static program analyses. Symbolic execution of a program means to run

```
if (x >=
    0)
{
   y=y-1;
}
else {
   y=y+1;
}
y=2*y;
```

(a) Program source code



(b) Symbolic execution tree for symbolic input values ($\texttt{x} : x_0, \texttt{y} : y_0$)

Fig. 1. Program and its symbolic execution tree

it with symbolic input values instead of concrete ones. Such a run results in a tree of symbolic execution traces, which covers all possible concrete executions.

Each node in a symbolic execution tree is annotated by its symbolic state. In the example shown in Fig. 1b, the root node is a branching node whose outgoing edges are annotated by their branch conditions. Here the symbolic execution splits into two branches: the left one for the case where the symbolic value x_0 is non-negative and the right one for a negative x_0. Both branches might be possible as we do not have any further information about the value of x_0. The path condition pc_i of a path sp_i is the conjunction of all its branch conditions and characterizes the symbolic execution path uniquely. As long as the program does not contain loops or method invocations, a path condition is a quantifier-free formula in first order logic.

From the above tree we can extract that in case of a non-negative input value for `x`, the program terminates in a final state in which the final value of `x` remains unchanged (i.e., x_0) while the final value of `y` is $2(y_0 - 1)$.

We fix the following notations as convention: Given a path i we refer to its path condition by pc_i and to the final value of a program variable $v \in Var$ by f_i^v. If we want to make explicit that the final value of a program variable v depends on the symbolic input value of a program variable u we pass it as an argument to f_i^v. For instance, $f_0^y(x_0, y_0) := 2(y_0 - 1)$ in case of the final value of y on the left branch.

In case of unbounded loops or unbounded recursive method calls a symbolic execution tree is no longer finite. We overcome this obstacle and achieve a finite representation by making use of specifications as proposed in [14]. This approach uses loop invariants and method contracts to describe the effect of loops and method calls. The basic idea is that loop invariants and method contracts contribute to path conditions and to the representation of the symbolic state. The approach has been implemented as a symbolic execution engine based on the verification system KeY [8], which we use as backend for the exploit generation presented in this paper.

Self-composition. Our exploit generation approach is derived from a logic-based formalization of noninterference using self-composition as introduced in [10,11], based on a direct semantic encoding of noninterference in a program logic. The Hoare triple $\{Pre\}$ `p` $\{Post\}$ is valid iff. the program `p` started in any initial state that satisfies formula *Pre* terminates, then formula *Post* must hold in the reached final state. The semantic definition of noninterference as given in Definition 3 requires the comparison of two program runs. The authors of [11] achieve this by copying the analyzed program and replacing all variables with fresh ones, such that the original and the copied program version do not share any memory. In more detail, let `p(l,h)` be the original program with $\texttt{l} \in L, \texttt{h} \in H$ being the only program variables occurring in program `p`. Further, let `p(l',h')` represent the copied program constructed from `p` by renaming variable l to l' and h to h'. Then

$$\{\texttt{l} \doteq \texttt{l'}\}\texttt{p(l,h); p(l',h')}\{\texttt{l} \doteq \texttt{l'}\}$$

is a direct formalization of noninterference. A major drawback of the formalization is that it requires program `p` to be analyzed twice. Several refinements have been presented since then to avoid the repeated program execution [5,24]. We make use of an approach based on symbolic execution. The fundamental idea is to execute the program symbolically only once and then to use the path conditions and symbolic states to construct a single first-order formula with the same meaning as the Hoare Triple. To express the noninterference property, we simply copy path conditions and symbolic values, replacing the symbolic input values with fresh copies.

3 Exploit Generation for Insecure Programs

3.1 Logic Characterization of Insecurity

Our objective is to generate an exploit for a given program `p` which demonstrates that `p` is insecure with respect to a specified information flow policy. The basic idea is that the exploit runs `p` twice and throws an exception if an unintended information flow is detected. The problem that needs to be solved is how to find the initial states for both runs such that an information leak can be observed.

To this end we construct a first-order formula which is satisfiable, if the program is insecure. The formula is constructed in such a way that any satisfying model can be directly used to construct the required initial states.

For noninterference, as defined in Def. 3, that formula is constructed as follows: Let $NI = (L, H)$ denote the noninterference policy with low variables L and high variables H. Each path sp_i $(i \in \{0, \ldots, n-1\})$ in the symbolic execution tree of `p` is uniquely characterized by its path condition $pc_i(L, H)$.

To represent two independent program runs, we create a copy of all program variables $Var' = \{v' \mid v \in Var\}$ and obtain the sets L' and H' as copies of L and H, i.e., $L' = \{l' \mid l \in L\}$ (analogously H'). Intuitively, the first run is performed using Var, while the second one uses the copy Var'. Both runs are independent

as they do not share any common memory. Then the *NI-insecurity formula*

$$\bigvee_{0\leq i\leq j<n}\Big(\big(\bigwedge_{l\in L,l'\in L'} l \doteq l'\big)\wedge pc_i(L,H)\wedge pc_j(L',H')\wedge\bigvee_{l\in L} f_i^l(L,H)\not\doteq f_j^l(L',H')\Big) \quad (1)$$

is satisfied iff. there is a model (i.e., state) σ assigning values to the program variables L, L', H, H' such that there exist two paths i, j ($i = j$ possible) with identical low level input, consistent path conditions (i.e., both paths can actually be taken), but for which the final value of at least one low level variable differs. In other words, the model σ assigns concrete values to L, L', H and H' such that $\mathtt{p}$ produces different low level output for two runs from initial states with identical low level input. (Formula (1) can be made more succinct by replacing L' with L and omitting the first conjunct, which states $L \doteq L'$.)

Example 1. The insecurity formula (1) for the example program from Fig. 1 and the NI policy $L = \{y\}, H = \{x\}$ becomes

$$\begin{aligned} & x_0 \geq 0 \wedge x_0' \geq 0 \wedge 2(y_0 - 1) \not\doteq 2(y_0 - 1) \\ \vee\ & x_0 \geq 0 \wedge x_0' < 0 \wedge 2(y_0 - 1) \not\doteq 2(y_0 + 1) \\ \vee\ & x_0 < 0 \wedge x_0' < 0 \wedge 2(y_0 + 1) \not\doteq 2(y_0 + 1) \end{aligned}$$

It is easy to see that the first and third disjunct are unsatisfiable, but the second disjunct is satisfiable, e.g., for the model $x_0 \mapsto 0, x_0' \mapsto -1, y_0 \mapsto 1$.

The NI-insecurity formula (1) can be rewritten into the equivalent formula

$$\bigvee_{l\in L}\bigvee_{0\leq i\leq j<n}\overbrace{\Big(\big(\bigwedge_{v\in L,v'\in L'} v \doteq v'\big)\wedge pc_i(L,H)\wedge pc_j(L',H')\wedge f_i^l(L,H)\not\doteq f_j^l(L',H')\Big)}^{Leak(H,L,l,i,j)} \quad (2)$$

which helps us later to incorporate declassification. The formula $Leak(H, L, l, i, j)$ allows to ascribe leaks to a specific target, i.e., it is satisfiable, if some information is leaked from the program variables in H to variable l.

3.2 Target Conditional Delimited Release

We extend the insecurity formula for noninterference (2) to delimited information release (DIR) [21]. In addition to the standard version of DIR, our policy describes not only what information might be released by using escape hatches, but it also allows to express under which condition and to whom (target) the information might be leaked.

Definition 5 (Target Conditional Delimited Release). *Given a program* $\mathtt{p}$ *with Var being the set of all variables occurring in* $\mathtt{p}$ *and a noninterference policy* $NI = (L, H)$. *A* Target Conditional Delimited Release (TCD) *policy* $(\mathcal{D}, NI)$ *is a set of* TCD specification triples *where each triple* $(e, C, T) \in \mathcal{D}$ *consists of*

- *an escape hatch expression (i.e., first order term) e over Var,*
- *a declassification condition formula C over Var and*
- *$T \subseteq L$, a set of program variables to which the specified escape hatch is allowed to be leaked.*

A program satisfies a given TCD policy $(\mathcal{D}, NI)$ if there is no information flow from H to L, except for the cases covered by a triple $(e, C, T) \in \mathcal{D}$ which allows the program to release the information captured by the escape hatch expression e to a location in T, if condition C is satisfied in the initial state of a program run.

Given a TCD policy $(\mathcal{D}, NI)$ and a program p. We give the insecurity formula for the case that $\mathcal{D} = \{(e, C, T)\}$ consists of a single TCD specification triple:

$$\bigvee_{l \in L} \left(Leak(H, L, l, i, j) \wedge ((l \in T \wedge C(Var) \wedge C(Var')) \rightarrow e(Var) = e(Var')) \right) \quad (3)$$

The formula coincides for locations $l \in L$ that are not allowed release targets (i.e., $l \notin T$) with the noninterference insecurity formula. Otherwise, the new second conjunct adds

$$C(Var) \wedge C(Var') \rightarrow e(Var) = e(Var')$$

as an additional restriction to the initial states for both runs, namely, that if both initial states satisfy the declassification condition C then they must also coincide on the value of the escape hatch expression. The rationale is that if there are two runs s.t. their respective initial state coincides on the low level input and on the escape hatches and if the final value for an allowed target differs, then more information than just the escape hatch must have been released.

The generalization of the above formula to more than one triple is straightforward and omitted for space reasons.

4 Exploit Generation Using Program Specifications

In this section we discuss how to use program specifications like loop invariants and method contracts to analyze and generate exploits for programs involving unbounded loops or recursive method calls. These specifications need to be user-provided at the moment, but work on automatic generation of specifications is ongoing. We focus on the noninterference analysis case, the extension to declassification is straightforward.

4.1 Loop Specification

The path conditions *pc* for a program p are computed by symbolic execution of p. The problem to solve is how to symbolically execute a loop. In case a fixed bound is known a priori the loop can simply be unwound, but this is impractical if the bound is large and not possible at all for unbounded loops.

In program verification, loops are handled by providing a loop specification. A loop specification $LS = (I, mod)$ consists of a loop invariant formula I and a set of program variables mod which contains all program variables the loop is allowed to modify. In [14] it is shown how to use such a specification for symbolic execution and we can simply reuse that option in our setting.

In the following we describe briefly how a loop specification is reflected in the NI- insecurity formula. Let $\mathtt{b}$ be the loop guard and $LS = (I, mod)$ the loop specification. The basic idea in [14] is that the loop specification describes the state after exiting the loop. This means, we can treat the loop as a black-box and continue execution after the loop in a state for which the variables mod that might have been modified by the loop are set to an unknown value. Unknown values are represented by the set of fresh symbolic variables V_{mod}. The only knowledge about the values of these variables is provided by the loop invariant and by the fact the loop guard $\mathtt{b}$ must be `false` (as we exited the loop).

Our insecurity formulas are always expressed as a constraint over the initial state. For instance, the final value f_i^l of variable `l` is given in terms of the initial symbolic values of the program variables. The same holds for the path conditions. We make this implicit weakest precondition computation here explicit for the loop guard and the invariant, i.e., I^{wp} is the weakest precondition of I computed in the state directly after the loop (similar for the loop guard).

For the sake of simplicity, we only show how to adapt $Leak(H, L, l, i, j)$ for the case that both paths i, j contain the same loop:

$$Leak(H, L, l, i, j) \equiv (\bigwedge_{v \in L} v = v') \wedge pc_i(V_S) \wedge pc_j(V'_S) \\ \wedge (I^{wp}(V_S) \wedge \neg \mathtt{b}^{wp}(V_S)) \wedge (I^{wp}(V'_S) \wedge \neg \mathtt{b}^{wp}(V'_S)) \wedge f_i^l(V_S) \neq f_j^l(V'_S) \quad (4)$$

where $b^{wp}(V_S)$ is the symbolic value of the guard after the loop, $V_S = Var \cup V_{mod}$. If one or both of paths i, j do not contain this loop, or have other loops, corresponding conjuncts are omitted or added accordingly.

Example 2. We illustrate formula (4). Consider the loop below with low variable `l` and high variable `h`. The loop specification is given as $(I : \mathtt{l} \geq 0, mod : \{\mathtt{l}\})$

```
1 l = h * h;
2 while (l > 0) { l = l - 1; }
3 l = l + h;
```

Let $\mathtt{l}_{mod}, \mathtt{l}'_{mod}$ be the fresh values representing the value of `l` directly after the loop. Computing the weakest precondition of the invariant gives us $\mathtt{l}_{mod} \geq 0$ and for the guard $\mathtt{l}_{mod} > 0$ for the first run (analog for the second run). The resulting formula is:

$$l = l' \wedge (l_{mod} \geq 0 \wedge \neg(l_{mod} > 0)) \wedge (l'_{mod} \geq 0 \wedge \neg(l'_{mod} > 0)) \wedge l_{mod} + h \neq l'_{mod} + h'$$

The formula is satisfiable for $\mathtt{l} = \mathtt{l}' = 10$, $\mathtt{l}_{mod} = \mathtt{l}'_{mod} = 0$, $\mathtt{h} = 1$ and $\mathtt{h}' = 2$. And actually the program is insecure. Removing the last statement would make

it secure and the formula unsatisfiable as the comparison of the final values would change to $\mathtt{l}_{mod} \neq \mathtt{l}'_{mod}$ which would be unsatisfiable.

4.2 Method Contracts

Let $\mathtt{m}$ denote a method. A contract $C_{\mathtt{m}}$ for $\mathtt{m}$ is a triple $(Pre_{\mathtt{m}}, Post_{\mathtt{m}}, Mod_{\mathtt{m}})$ with precondition $Pre_{\mathtt{m}}$, postcondition $Post_{\mathtt{m}}$ and modifies (or assignable) clause $Mod_{\mathtt{m}}$, which enumerates all program variables that $\mathtt{m}$ is allowed to change.

A method satisfies its contract, if it ensures that when invoked in a state for which the precondition is satisfied, then in the reached final state the postcondition holds and at most the program variables (locations) listed in the assignable clause have been modified.

Analysing Noninterference w.r.t. to a Precondition. Given a method $\mathtt{m}$ with contract $C_{\mathtt{m}}$. We want to analyze whether $\mathtt{m}$ respects its noninterference policy $NI = (L, H)$ under the condition that $\mathtt{m}$ is only invoked in states satisfying its precondition $Pre_{\mathtt{m}}$. Changing the noninterference formula (2) is easy and only requires adding a restriction to the initial states requiring them to satisfy the method's precondition:

$$Leak(H, L, l, i, j) \wedge Pre_{\mathtt{m}}(L, H) \wedge Pre_{\mathtt{m}}(L', H') \tag{5}$$

Analyzing Programs for Noninterference using Method Contracts. A similar problem as for loops manifests itself when symbolically executing a program which invokes one or more methods. One solution is to replace the method invocation by the body of the invoked method. If the methods are small this is a viable solution, but it is impractical if the invoked method is complex and is even impossible for recursive methods without a fixed maximal recursion depth.

This problem is solved in [14] by using method contracts in a similar way loop specifications have been used. Instead of a loop invariant, the pre- and postconditions become parts of the path conditions. The modifies clause gives again rise to fresh variables used to represent the symbolic value of the program variables that might have been changed as side-effect of the method invocation.

Let $\mathtt{m}$ be the method that is analyzed for secure information flow and which invokes a method $\mathtt{n}$. The method contract of $\mathtt{n}$ is given as $(Pre_{\mathtt{n}}, Post_{\mathtt{n}}, Mod_{\mathtt{n}})$. For the case that both two paths i, j contain one method call for $\mathtt{n}$ we get:

$$\begin{aligned} Leak(H, L, l, i, j) \equiv (\bigwedge_{v \in L} v = v') \wedge pc_i(V_S) \wedge pc_j(V'_S) \wedge (Pre^{wp}_{\mathtt{n}}(V_S) \wedge Post^{wp}_{\mathtt{n}}(V_S)) \\ \wedge (Pre^{wp}_{\mathtt{n}}(V'_S) \wedge Post^{wp}_{\mathtt{n}}(V'_S)) \wedge f^l_i(V_S) \neq f^l_j(V'_S) \end{aligned} \tag{6}$$

where V_S is the set of program variables extended by the newly introduced variables resulting from the modifies clause of method $\mathtt{n}$ and $Pre^{wp}_{\mathtt{n}}, Post^{wp}_{\mathtt{n}}$ are the weakest preconditions of $Pre_{\mathtt{n}}, Post_{\mathtt{n}}$ computed directly before (resp. after) the method invocation. The general case is similar to loops.

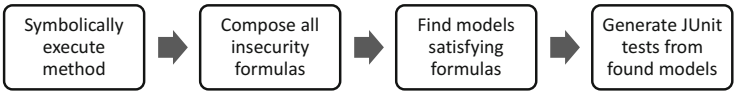


Fig. 2. Exploit Generation by KEG

4.3 General Observations and Remarks

Using loop specifications or method contracts has one major drawback, namely, that not all models of a formula give rise to an actual information leak, or even worse, the insecurity formula of a secure program might become satisfiable. This case does not effect the soundness, but triggers false warnings. The reason is that the specifications might be too weak and allow behaviours that are not possible in the actual program. These false warnings can be filtered out by actually running the generated exploit. If the exploit fails to demonstrate the information leak, we know that our model was a spurious one. We can even start a feedback loop with a conflict clause which rules out the previously found model.

On the other side if loop or method specifications are not just too weak, but wrong in the sense that they exclude existing behaviour, leaks might not be detected. This can be avoided by verifying the specifications using a program verification tool. As we are concerned with bug detection and not verification, this case is not too bad as we do not claim to find all bugs.

5 Implementation and Experiments

5.1 The KeY Exploit Generation Tool

We implemented our approach as a tool called *KeY Exploit Generation (KEG)*[1] based on the verification system KeY for Java [8]. KEG uses KeY as symbolic execution (SE) engine, which supports method and loop specifications to achieve a finite SE tree. The SMT solver Z3 is used to find models for the insecurity formulas. KEG is able to deal with object types and arrays (to some extent).

Fig. 2 outlines KEG's work-flow. As starting point it serves a Java method `m` which is analysed for secure information flow w.r.t. a given information flow specification. First, method `m` is symbolically executed (using KeY) to obtain the SE tree with the method's path conditions and the final symbolic values of the program locations modified by `m`. Using this information the insecurity formulas are generated and given to a model finder (in our case the SMT solver Z3). If a model for the insecurity formula has been found, the model is used to determine the initial states of two runs which exhibit a forbidden information flow. The generated exploit sets then up two runs (one for each initial state) and inspects the reached final states to detect a leak. KEG outputs the exploited program as a JUnit test to be included into a regression test suite.

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

5.2 Exploit Generation Using a Simple Example

We explain KEG using the simple example shown below:

```
1 public class Simple {
2     public int l; private int x, y;
3     /*! l | x y ; !*/
4
5     /*@ escapes (x*y) \to l \if x>-1; @*/
6     public void magic() { if(x>0) { l=x*y; } else { l=0; } }
7 }
```

Class `Simple` contains three integer typed fields `l`, `x` and `y` as well as a method called `magic()` which assigns a value to `l` depending on the sign of field `x`. The information flow policy is annotated as special comment types. Line 3 is a class level specification and forbids any information flow from `x` and `y` to `l`. Hence, here `x` and `y` are high variables and `l` is a low variable. However, this strict noninterference policy is relaxed in line 5 for method `magic()` by providing a target conditional release specification consisting of an escape hatch (`x*y`), the target `l` and the condition `x>-1`.

Running KEG on the above example produces a symbolic execution tree consisting of two paths; one for each branch of the conditional statement. KEG generates for each pair of these paths the corresponding insecurity formulas and passes them on to an SMT solver. Of the three generated insecurity formulas only one is satisfiable and Z3 provides a model:

Insecurity Formula	Model

`(let ((a!1 (not (and (> self_x_1 (- 1)) (> self_x_2 (- 1))))))`
`  (and (>= self_x_1 1)  (<= self_x_2 0)`
`  (or (not (= self_x_1 self_x_2)) (not (= self_y_1 self_y_2)))`
`  (= self_l_1 self_l_2) (not (= (* self_y_1 self_x_1) 0))`
`  (or a!1 (= (* self_x_1 self_y_1) (* self_x_2 self_y_2)))))`
	`self_x_1 : 1`
	`self_x_2 : -1`
	`self_y_1 : -1`
	`self_y_2 : 1`
	`self_l_1 : 0`
	`self_l_2 : 0`

The formula comparing two runs which take different branches of the conditional statement and thus leak the sign of field `x`. KEG generates exactly one exploit, which is output as a well-structured and human readable JUnit test.

5.3 Experiments

We performed a number of small experiments[2] for a first evaluation of our approach. Table 1 shows the aggregated results. All experiments were done on an Intel Core i7-4702HQ processor with JVM setting -Xmx4096m.

Concerning the runtime performance: A significant amount is spent for parsing the program, this can be reduced by parser optimizations, e.g., by using a hand-coded version instead of a generated parser. Model finding time can be optimized by performing simple techniques like symmetry reduction, learning and caching, all of which have not yet been implemented. Another factor is the programming language Java whose optimizations are performed at runtime and, hence, code run only a few times will not be optimized at all.

[2] www.se.tu-darmstadt.de/fileadmin/user_upload/Group_SE/Tools/KEG/experiments.zip

Table 1. Case studies statistics

File name	Analyzed Method	#L/MI	Policy (NI/D)	S/I	T_L (ms)	T_{SE} (ms)	T_{MF} (ms)	T_{Tot} (ms)	#GE/FW
Mul	product	0 / 0	D	I	4187	847	1188	6266	1 / 0
Mul_StrongLI	product	1 / 0	D	I	4275	1746	1211	7274	1 / 0
Mul_WeakLI	product	1 / 0	D	I	4214	1909	1293	7463	2 / 1
Mul_WrongLI	product	1 / 0	D	I	4397	1678	1169	7285	0 / 0
Comp_StrongMC	doWork	0 / 1	NI	I	4181	1491	2278	7995	3 / 0
Comp_WeakMC	doWork	0 / 1	NI	I	4217	1383	2417	8065	3 / 3
Comp_WrongMC	doWork	0 / 1	NI	I	4182	1395	2275	7887	0 / 0
Company	calculate	1 / 1	NI	I	4283	2496	1990	8816	3 / 0
ExpList	magic	0 / 0	NI	I	4178	1911	2535	8668	1 / 0
ExpLinkedList	magic	0 / 4	NI	I	4229	4690	6564	15526	2 / 0
ExpArrayList	magic	0 / 5	NI	I	4230	8975	11505	24752	3 / 0
ArrMax	findMax	1 / 0	NI	I	4215	3584	963	8804	1 / 0
ArrSearch	search	1 / 0	D	S	4199	2934	2400	9568	0 / 0

#(L/MI/GE/FW): nr of **L**oops/**M**ethod **I**nvocations/**G**enerated **E**xploits/**F**alse **W**arnings
NI/D: **N**on-**I**nterference/**D**eclassification, S/I: **S**ecure/**I**nsecure
T_X: Time for **L**oading/**S**ymbolic **E**xecution/**M**odel **F**inding/**Tot**al

A few observations concerning some of the concrete case studies: For the examples *Mul* and *Comp*, we analyzed also the effect of loop specifications resp. method specifications in case of strong, weak and wrong specifications (*file-name*_Strong/Weak/Wrong_LI/MC). As expected in case of sufficiently strong specifications, all insecure paths could be identified and corresponding exploits have been generated. Weak specifications over-approximated possible behaviour leading to false warnings, while wrong specifications excluded actual behaviours and missed existings leaks. The analysis of method `search` in class *ArrSearch* identified the method correctly as secure with respect to the specified declassification policy and generated no exploits.

6 Related Work

Our approach to exploit generation is based on self-composition [6,10,11]. The paper [11] addresses also declassification. Its authors observe that in their formalization it is possible to express and verify that a program is insecure. Our formalization of insecurity uses this observation. Exploit generation (extraction of models) in our paper owes to techniques developed for automatic test generation. In particular, we build upon work presented in [12,16], where symbolic execution is used as a means to generate test cases for functional properties.

Logic-based approaches such as [7,23] are fully precise and at the same time can flexibly express various information flow properties beyond the policies presented in this paper. The verification process is not fully automatic, however, and non-trivial interactions with the theorem prover are required. In [19] higher order logic is used to express information flow properties for object-oriented programs, which is highly expressive, but poses also a high demand on user interaction.

Pairs of symbolic execution paths instead of standard self-composition have been independently used in [20] to check programs for noninterference. However, the author is only concerned with checking noninterference, but does not support declassification. Unbounded loops and recursive methods are not addressed.

In [25], leaks are inferred automatically and expressed in a human-readable security policy language helping programmers to decide whether the program is secure or not, however it can not give concrete counterexamples that could suggest further corrections. Counterexamples can be used not only to generate executable exploits as in our approach, but also to refine declassification policies quantifying the leakage [1,3]. However, both above approaches do not provide a solution for unbounded loops and recursions.

ENCoVer [2] uses epistemic logic and makes use of symbolic execution (concolic testing) to check noninterference for Java programs. In [17], the authors proposed a tool which checks that a C program is secure w.r.t. noninterference. It transforms the original program and makes use of dynamic symbolic execution to analyze the program's information flow. Both tools check loops and recursive method invocations only up to a fixed depth.

Type-based approaches to information flow like [15,18,21,26] or those based on dependency graphs [13] distinguish themselves by their high performance and ability to check large systems. Common drawbacks are lack of precision and resulting false warnings and/or restrictions on the syntactic form of a program.

None of the logic-based and type-based approaches to noninterference analysis mentioned above does generate exploits from a failed proof or analysis. Our work does not intend to replace their approaches, but to be used complementary.

7 Conclusion

We presented a novel approach for automatically detecting information flow leaks in object-oriented imperative programs. Exploits are generated based on satisfying models of *insecurity formulas* and output as tests so that they can easily be integrated into regression test collections. We also showed how program specifications such as loop invariants and method contracts can be used to overcome the obstacle of an infinite symbolic execution tree in case of unbounded program structures. We have built a prototypical tool (KEG) based on our approach that handles sequential Java programs and we applied it to a number of case studies.

We plan to integrate KEG with the abstraction framework presented in [27] which allows us to automatically generate loop invariant and method contracts to avoid the need for user-provided specifications.

References

1. Backes, M., Kopf, B., Rybalchenko, A.: Automatic discovery and quantification of information leaks. In: Proc. of the 30th IEEE Symp. on Security and Privacy, pp. 141–153. SP 2009, IEEE CS (2009)
2. Balliu, M., Dam, M., Le Guernic, G.: ENCoVer: symbolic exploration for information flow security. In: 25th IEEE Computer Security Foundations Symposium, pp. 30–44. IEEE CS (2012)
3. Banerjee, A., Giacobazzi, R., Mastroeni, I.: What you lose is what you leak: Information leakage in declassification policies. ENTCS **173**, 47–66 (2007)
4. Banerjee, A., Naumann, D.A.: Stack-based Access Control and Secure Information Flow. J. Funct. Program. **15**(2), 131–177 (2005)
5. Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 200–214. Springer, Heidelberg (2011)
6. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. In: Proc. of the 17th IEEE Workshop on Computer Security Foundations, pp. 100–114. CSFW 2004, IEEE CS (2004)
7. 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)
8. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach. LNCS (LNAI), vol. 4334. Springer, Heidelberg (2007)
9. Cohen, E.S.: Information Transmission in Sequential Programs. Foundations of Secure Computation, pp. 297–335 (1978)
10. Darvas, A., 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, ACM SIGPLAN and GI FoMSESS (2003)
11. Darvas, A., Hähnle, R., Sands, D.: A theorem proving approach to analysis of secure information flow. In: Hutter, D., Ullmann, M. (eds.) SPC 2005. LNCS, vol. 3450, pp. 193–209. Springer, Heidelberg (2005)
12. Engel, C., Hähnle, R.: Generating unit tests from formal proofs. In: Gurevich, Y., Meyer, B. (eds.) TAP 2007. LNCS, vol. 4454, pp. 169–188. Springer, Heidelberg (2007)
13. Graf, J., Hecker, M., Mohr, M.: Using JOANA for information flow control in java programs - a practical guide. In: Proc. of the 6th Working Conf. on Programming Languages, pp. 123–138. LNI 215, Springer (February 2013)
14. 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)
15. Hunt, S., Sands, D.: On flow-sensitive security types. In: ACM SIGPLAN Notices, vol. 41, pp. 79–90. ACM (2006)
16. King, J.C.: Symbolic Execution and Program Testing. Commun. ACM **19**(7), 385–394 (1976)
17. Milushev, D., Beck, W., Clarke, D.: Noninterference via symbolic execution. In: Giese, H., Rosu, G. (eds.) FORTE 2012 and FMOODS 2012. LNCS, vol. 7273, pp. 152–168. Springer, Heidelberg (2012)
18. Myers, A.C.: JFlow: practical mostly-static information flow control. In: Proc. of 26th ACM Symp. on Principles of Programming Languages, pp. 228–241 (1999)

19. Nanevski, A., Banerjee, A., Garg, D.: Verification of information flow and access control policies with dependent types. In: Proc. of the 2011 IEEE Symp. on Security and Privacy, pp. 165–179. SP 2011, IEEE CS (2011)
20. Phan, Q.S.: Self-composition by symbolic execution. In: Jones, A.V., Ng, N. (eds.) Imperial College Computing Student Workshop. OASIcs, vol. 35, pp. 95–102. Schloss Dagstuhl (2013)
21. Sabelfeld, A., Myers, A.C.: A model for delimited information release. In: Futatsugi, K., Mizoguchi, F., Yonezaki, N. (eds.) ISSS 2003. LNCS, vol. 3233, pp. 174–191. Springer, Heidelberg (2004)
22. Sabelfeld, A., Sands, D.: Declassification: Dimensions and Principles. Journal of Computer Security **17**(5), 517–548 (2009)
23. 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)
24. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 352–367. Springer, Heidelberg (2005)
25. Vaughan, J.A., Chong, S.: Inference of expressive declassification policies. In: Proc. of the 2011 IEEE Symp. on Security and Privacy, pp. 180–195. IEEE CS (2011)
26. Volpano, D., Irvine, C., Smith, G.: A Sound Type System for Secure Flow Analysis. Journal of Computer Security **4**(2), 167–187 (1996)
27. Wasser, N., Bubel, R.: A theorem prover backed approach to array abstraction. In: Proc. of VSL 2014 – WING Workshop (2014)