Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Automatic detection and demonstrator generation for information flow leaks in object-oriented programs ☆

Quoc Huy Do *, Richard Bubel, Reiner Hähnle

TU Darmstadt, Dept. of Computer Science, Darmstadt, Germany

ARTICLE INFO

Article history:
Available online 13 December 2016

Keywords:
Test generation
Symbolic execution
Information flow
Declassification
Information erasure
Bug finding

ABSTRACT

We present a method to generate automatically exploits for information flow leaks in object-oriented programs. The goal, similar to white-box test generation, is to automatically produce executable, reusable test cases that challenge a given information flow policy with a very high degree of guaranteed coverage. Our approach combines self-composition and symbolic execution to create an insecurity formula for a given program and information flow policy. Satisfiability of this formula signifies the presence of information leaks and permits to use model generation for creating exploits. We support different kinds of information flow policies like noninterference, delimited information release, and information erasure. A prototypic tool implementation for Java programs of our approach is available. It generates exploits in the form of self-contained, executable JUnit tests. We evaluate our method and tool based on a set of micro-benchmarks and a case-study on e-voting.
© 2016 Elsevier Ltd. All rights reserved.

1. Introduction

Ensuring that programs do not leak unintended information is essential to establish confidence in the ability of an IT system not to put the security and privacy of its users at risk. In the past decade, much theoretical and practical work on information flow analysis of programs was produced (for example Askarov et al., 2015; Banerjee and Naumann, 2005; Barthe et al., 2004, 2011; Dam et al., 2010; Darvas et al., 2005; Devriese and Piessens, 2010; Hunt and Sands, 2006; Myers, 1999; Sabelfeld and Myers, 2004; Scheben and Schmitt, 2012; Volpano et al., 1996). Its focus is to ensure that an outside agent with well-defined properties cannot infer secret inputs by observing (and initiating) several runs of a program.

Much research on information flow analysis aims at *statically* proving that a program does not leak any unintended information. One can distinguish type-based (e.g. Banerjee and Naumann, 2005; Hunt and Sands, 2006; Myers, 1999; Sabelfeld and Myers, 2004; Volpano et al., 1996) and logic-based (e.g. Barthe et al., 2004; Darvas et al., 2005; Scheben and Schmitt, 2012) approaches. The former tends to be automatic, but imprecise, while the latter tends to be precise, but requires expert interaction.

Precision and automation at the same time can be achieved *dynamically*, for a concrete program run. Security monitors (e.g. Askarov et al., 2015; Dam et al., 2010) raise a warning as soon as a program violates a given security policy and try to contain the leak. Secure multi-execution ((Devriese and Piessens, 2010) and several follow-up papers) determines whether a given concrete run might violate a policy.

In this paper we connect the static and dynamic view: by static analysis we produce test cases, which we call *exploits*, that are guaranteed to violate a given security policy. Such

☆ The work has partially been funded by the DFG priority program 1496 "'Reliably Secure Software Systems'".
* *Corresponding author.*
E-mail address: do@cs.tu-darmstadt.de (Q.H. Do).
http://dx.doi.org/10.1016/j.cose.2016.12.002
0167-4048/© 2016 Elsevier Ltd. All rights reserved.

exploits can then be used independently for system testing, regression testing, documentation, etc. We take a similar view here as in functional verification: a security policy can be seen as a requirements specification. Its violation is a software fault witnessed by a test case, i.e. exploit.

In functional verification it is well-known that static verification is not a replacement for testing, but both techniques complement each other (Beckert and Hähnle, 2014): static verification gives strong guarantees for the modeled part of a system, while testing is incomplete, but validates actual executables and can also find problems in the underlying runtime environment or hardware platform. In our work we intend to achieve similar goals as in white-box software test generation frameworks (Albert et al., 2010; de Halleux and Tillmann, 2008; Engel and Hähnle, 2007).

- Under the assumption that a sound and complete specification is provided, no further user interaction is required and the approach is automatic.
- Completeness is achieved only in specific cases (e.g., no loops present or else strong loop invariants are supplied), however, strong and precise coverage guarantees can be given.
- Like test cases, exploits can be used to validate a program in its actual runtime environment.
- Like test cases, exploits become part of a library that is regularly executed to protect against regression. They are useful even after changes were made.
- Like test cases, exploits can serve as a documentation and illustration of intended system behavior.

Unlike monitoring and multi-execution, no special runtime infrastructure is required, because we generate exploits in the form of self-contained JUnit tests. These run the program under test multiple times with the generated input in order to produce a security violation. In addition, the generated exploits come with statically determined coverage guarantees.

A huge problem in software testing is the creation of oracles (Barr et al., 2015) that tell whether a test succeeded. Among the problems are i) missing or insufficient specifications, ii) complexity of general functional specifications which might contain quantifications over all objects and similar and iii) possible unintended side-effects of specification code (runtime exceptions and similar). Even in automatic test generation, these must often be supplied manually. But information flow policies (see Sabelfeld and Sands (2009) for an overview) can usually be expressed in a uniform manner for any given program. Hence, it is possible to generate test oracles automatically from them. In fact, our approach goes one step further in being *oracle-sensitive*: only such exploits that violate a given policy are generated at all.

Like other white-box software test generation frameworks (Albert et al., 2010; de Halleux and Tillmann, 2008; Engel and Hähnle, 2007; King, 1976), our method is based on symbolic execution of the target program. In our case, it is embedded (Gladisch, 2008) into the program logic of the deductive verification framework KeY (Beckert et al., 2007), which is also the basis for the acronym of our tool KEG (for KeY Exploit Generator). The relational nature of information flow (two runs of a program must be compared) is captured by the technique of self-composition (first introduced in Darvas et al. (2003); the name self-composition was coined in Barthe et al. (2004)). The result is an *insecurity formula* for a given information flow policy that is satisfiable if and only if the policy is violated. Model generation with the help of SMT solvers (de Moura and Bjørner, 2008) yields the input data for the exploit. In addition to standard information flow policies like noninterference, we also support relativized properties that tend to be used in practice, including delimited information release and information erasure (Sabelfeld and Sands, 2009).

The paper is structured as follows. Section 2 introduces basic notions and techniques. Section 3 explains the logic formalization of security policies. In Section 4 we discuss the analysis of loops and method invocations, needed to deal with realistic programs. Section 5 presents our tool KEG and gives a proof-of-concept with micro benchmarks. Our method and its implementation KEG are evaluated with the help of a larger case study (an e-voting program) in Section 6. We compare our work with others in Section 7 and conclude with Section 8.

This paper is an extended version of Do et al. (2015). The main new aspects are a generalization of the encoding of noninterference to include information erasure (Section 3.2) and the e-voting case study in Section 6 which is completely new. Minor additions concern micro benchmarks (Section 5.3) and a detailed description of the structure of the generated JUnit tests (Section 5.2).

2. Background

2.1. Information flow policies

Before we can analyze that a program does not leak confidential information, we need to define the *security requirements*. This has two aspects: the *security level* of each program location (i.e. program variables and fields) as well as an *information flow policy* defining whether and what kind of information may flow between program locations with a different security level. We recall the definitions of two well-known information flow policies supported by our approach.

Noninterference. Noninterference (Cohen, 1978; Volpano et al., 1996) is the strongest possible information flow policy. It typically involves two security levels (high/confidential vs. low/public) and completely prohibits any information flow from program locations containing confidential information) to publicly observable program locations. The opposite direction is allowed. In our work we consider only deterministic programs. In this case, noninterference can be formalized by comparing two program runs:

Definition 1. (Noninterference – Informal). *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.*

In other words the final value of low variables is solely determined by the initial value of low variables and does not depend on the initial values of high variables.

We define some basic notions required to formalize information flow policies. Let p denote a program and *Var* the set of all program variables[1] of p.

Definition 2. (Program State). *A program state σ maps each program variable $v \in Var$ of type T (write $v : T$) 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 p is denoted as $States_p$.

We define coincidence of program states relative to a set of program variables:

Definition 3. (State Coincidence). *Given a set of program variables V and two states $\sigma^1, \sigma^2 \in States_p$, we write $\sigma^1 \simeq_V \sigma^2$ if and only if 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 p is a possibly infinite sequence of program states $\tau = \sigma_0\sigma_1\sigma_2\cdots$ produced by starting p in state σ_0. In this paper, we concern ourselves only with terminating[2] programs, consequently, all of our execution traces are finite. Then the big-step semantics is defined as follows: let *X* be a concrete execution of a program p defined by a trace τ^X. We represent X by a pair $\langle \sigma^X, \sigma^X_{out} \rangle$, where $\sigma^X \in States_p$ is the first state of τ^X and $\sigma^X_{out} \in States_p$ is the last. The set of all possible concrete executions of p is denoted as Exc_p. We can now formally define noninterference for two security levels low and high:

Definition 4. (Noninterference). *Given a program p over variables Var and a* noninterference policy $NI = H \not\rightsquigarrow L$ *where $L \dot\cup H = Var$ such that L contains the low variables and H the high variables, program p has secure information flow with respect to NI if and only if for all concrete executions $X, Y \in Exc_p$ it holds that if $\sigma^X \simeq_L \sigma^Y$ then $\sigma^X_{out} \simeq_L \sigma^Y_{out}$.*

Example 1. The program:

```
if (h > 0) {l = 2;}
```

with high variable *h* and low variable l is insecure as it does not satisfy the noninterference property for the policy $NI = \{h\} \not\rightsquigarrow \{l\}$. Given two initial states σ^1 with $\sigma^1(h) = 5$, $\sigma^1(l) = 0$ and σ^2 with $\sigma^2(h) = -5$, $\sigma^2(l) = 0$, respectively. They satisfy $\sigma^1 \simeq_{\{l\}} \sigma^2$, but in the final states we have $\sigma^1_{out}(l) = 2 \neq \sigma^2_{out}(l) = 0$.

Declassification. In practice noninterference is too restrictive. For instance, a program that authenticates users with their login password leaks the information whether an entered password is correct, or take a database that may be queried for aggregated values like the average salary, but not for the income of an individual person.

Declassification is a class of information flow policies that allows one to express that some precisely specified confidential information may be leaked. The paper Sabelfeld and Sands (2009) provides an extensive survey of declassification approaches. Here we consider *delimited information release* as introduced in Sabelfeld and Myers (2004). Delimited information release is a declassification policy which allows one 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:

$$\frac{\sum_{e \in Person} salary(e)}{|Person|}$$

can be used to declassify the average of the income of all persons in a database. The formal definition of delimited information release extends Definition 4. Both definitions coincide for trivial escape hatches such as $e = true$.

Definition 5. (Delimited Information Release). *Given a program p over variables Var and a* delimited information release policy *$Decl = (L, H, E)$ with L, H as before and E denoting a set of escape hatch expressions, program p has secure information flow with respect to Decl if and only if 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.*

Example 2. Consider again the program from Example 1:

```
if (h > 0) {l = 2;}
```

Given the delimited information release policy $Decl = (\{l\}, \{h\}, \{h > 0\})$ where the escape hatch allows the sign of h to be leaked, the counter example for noninterference from Example 1 is no longer a counter example as:

$$[\![h > 0]\!]_{\sigma_1} = true \neq false = [\![h > 0]\!]_{\sigma_2}.$$

In fact the program is secure for the given policy, as the decision whether l may be altered is only based on the guard. The same policy for the program:

```
if (h > 0) {l = h;}
```

is insecure as can be demonstrated by the following counter example: Given initial states σ^1 with $\sigma^1(h) = 3$, $\sigma^1(l) = 0$ and σ^2 with $\sigma^2(h) = 5$, $\sigma^2(l) = 0$ we observe (i) both states coincide on the value of the low variable l; and (ii) they evaluate the escape hatch expression to the same value $[\![h > 0]\!]_{\sigma_1} = [\![h > 0]\!]_{\sigma_2} = true$, but their final states differ on the value of l: $\sigma^1_{out}(l) = 3 \neq 5 = \sigma^2_{out}(l)$.

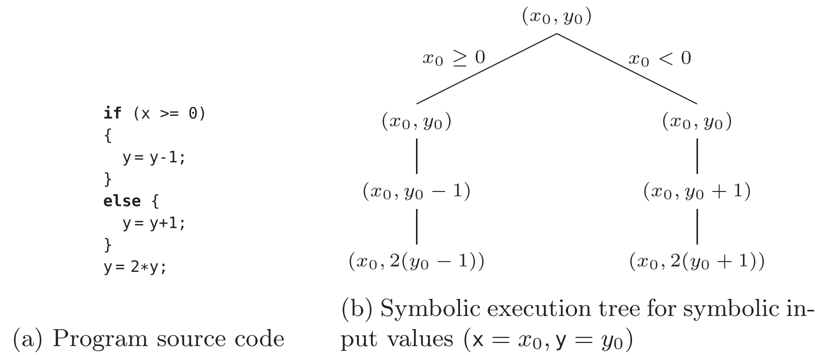
In Section 3.2 we will further generalize the delimited information release policy to include information erasure.

2.2. *Logic-based information flow analysis*

Symbolic Execution. Symbolic Execution (King, 1976) is a versatile technique used for various static program analyses.

[1] To keep the presentation manageable, in the formal definitions we mention only variables, however, our implementation works also for fields, including reference types.

[2] This is purely in the interest of scoping. Our approach is not principally limited to terminating programs.



(a) Program source code

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

Fig. 1 – A program and its symbolic execution tree.

Symbolic execution of a program means to run it with symbolic input values instead of concrete ones. Such a run results in a tree of symbolic execution traces, which cover all possible concrete executions.

Each node in a symbolic execution tree is annotated by its symbolic state given as a tuple of symbolic expressions, each tuple element corresponding to a program variable. 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 nonnegative and the right one for a negative x_0. Both branches might be taken as we do not have any further information about the value of x_0. The *path condition* of a path 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 tree in Fig. 1b we can extract that in case of a nonnegative 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 make some notational conventions: Given path *i* we refer to its path condition by pc_i and to the final value of a program variable v on that path with 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 we pass it as an argument to f_i^v. For instance, if the left branch is numbered with 0, then the final value of y on that branch is $f_0^y(x_0, y_0) = 2(y_0 - 1)$.

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 Hentschel et al. (2014). That 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 modification of the symbolic state. This is realized in the symbolic execution engine of the verification system KeY (Beckert et al., 2007), which we use as implementation basis of KEG.

Self-composition. Our exploit generation approach is derived from a logic-based formalization of noninterference using self-composition as introduced in Darvas et al. (2003, 2005), based on a direct semantic encoding of noninterference in a program logic. The Hoare triple $\{Pre\}\mathtt{p}\{Post\}$ characterizes that, whenever the program p, started in an initial state satisfying *Pre*, terminates, then *Post* must hold in the final state reached. Noninterference as given in Definition 4 requires the comparison of two program runs. In Darvas et al. (2005) this is achieved by copying the program p and replacing all its variables with fresh copues, such that the original and the copied version do not share any memory. Concretely, let without loss of generality $\mathtt{l} \in L$, $\mathtt{h} \in H$ be the only variables of $\mathtt{p} = \mathtt{p}(\mathtt{l}, \mathtt{h})$. Further, let $\mathtt{p}(\mathtt{l}', \mathtt{h}')$ represent the copied program constructed from p by renaming variable *l* to *l′* and *h* to *h′*. Then:

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

is a direct formalization of noninterference. A major drawback of this formalization is that it requires program p to be analyzed twice. Several refinements have been presented to avoid the repeated execution (Barthe et al., 2011; Terauchi and Aiken, 2005). Here we use a different approach to that problem 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 that has same meaning as the Hoare triple. To express the noninterference property, it is then sufficient to copy path conditions and symbolic values, replacing the symbolic input values with fresh copies. The technical details are given in the following section.

3. Exploit generation for insecure programs

We describe the formalization in logic for various information flow policies. Given a complete symbolic execution tree for a program and an information flow policy, we construct formulas that are unsatisfiable when the program is secure and satisfiable if the policy can be violated by some inputs. This approach permits to use an SMT solver or model finder to search for satisfying models from which one can then read off concrete input states for two program runs that demonstrate a violation of the given policy, see also Fig. 2.

3.1. *Logic characterization of insecurity*

First we show how to construct a formula that characterizes noninterference (Definition 4) from a complete symbolic

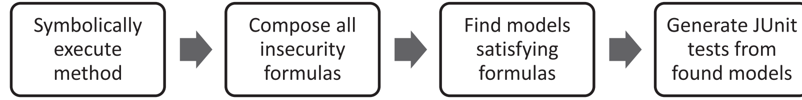


Fig. 2 – Exploit Generation by KEG.

execution tree for program p with paths $i \in \{0,\ldots,n-1\}$. Let $pc_i(L, H)$ be the path condition that uniquely determines path i. Let $NI = H \not\rightsquigarrow L$ be the noninterference policy with low variables L and high variables H.

To represent two independent program runs, we create a copy of all program variables $Var' = \{v' | v \in Var\}$ with the notational convention that v' always refers to the copy of v. Now we obtain the sets L' and H' as copies of L and H, i.e., $L' = \{l' | l \in L\}$ (analogously H'). To refer to the initial (symbolic) value of a program variable v, instead of introducing a new constant symbol v_0, we simply use the program variable v itself. 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:*

$$\bigvee_{0\le i\le j<n}\left(\bigwedge_{l\in L} l \doteq l' \wedge pc_i(L,H) \wedge pc_j(L',H') \wedge \bigvee_{l\in L} f_i^l(L,H) \ne f_j^l(L',H')\right) \quad (1)$$

is satisfied if and only if there is a model (i.e., concrete state) σ assigning values to the program variables Var, Var' such that

- the input values of the low variables L coincide with the values of their copies in L',
- there are two paths i, j ($i = j$ possible) with consistent path conditions (i.e., both paths can actually be taken), but
- for which the final value of at least one low variable differs.

In other words, the model σ assigns concrete values to Var and Var' such that p produces different low level outputs for two runs from initial states with identical low input.

Example 3. The insecurity formula (1) for the example program from Fig. 1 and the NI policy $\{y\} \not\rightsquigarrow \{x\}$ is:

$$\begin{aligned} & y_0 \doteq y_0' \wedge x_0 \ge 0 \wedge x_0' \ge 0 \wedge 2(y_0-1) \ne 2(y_0'-1) \\ \vee\ & y_0 \doteq y_0' \wedge x_0 \ge 0 \wedge x_0' < 0 \wedge 2(y_0-1) \ne 2(y_0'+1) \\ \vee\ & y_0 \doteq y_0' \wedge x_0 < 0 \wedge x_0' < 0 \wedge 2(y_0+1) \ne 2(y_0'+1) \end{aligned}$$

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

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

$$\bigvee_{l\in L}\bigvee_{0\le i\le j<n}\overbrace{\left(\left(\bigwedge_{l\in L} l \doteq l'\right) \wedge pc_i(L,H) \wedge pc_j(L',H') \wedge f_i^l(L,H) \ne f_j^l(L',H')\right)}^{Leak^{NI}(H,L,l,i,j)} \quad (2)$$

This formulation will be easier to incorporate declassification. The intuition behind the formula $Leak^{NI}(H, L, l, i, j)$ is that it allows us 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.

The copy of the low level variables is actually not needed (as we require their equality for all models), so formulas (1) and (2) can be made more succinct by replacing L' with L and omitting the first conjunct, which states $L \doteq L'$. In the future we will tacitly perform this simplification. Then the first disjunct in Example 3 becomes $x_0 \ge 0 \wedge x_0' \ge 0 \wedge 2(y_0-1) \ne 2(y_0-1)$.

3.2. *Generalized noninterference policy*

Sometimes it is not sufficient to simply ensure that no information is leaked, but one wants also to guarantee that secret data are not kept longer than needed, because of legal reasons or to make data dumps (initiated by an attacker) less useful. *Information erasure* is for a desired property for cryptographic devices (secret keys must be erased after usage), online transactions (credit card information must be erased after the transaction is completed), e-voting (all data connecting voter and ballot must be erased after the result has been published), etc. Information erasure policies have been presented in Del Tedesco et al. (2011) and Hunt and Sands (2008).

Listing 1.1: Ticket vending machine

```
1 void buy() {
2   charge(ticketCost, ccNumber);
3   log();
4   ccNumber = 0;
5 }
```

Example 4. Consider a simple ticket vending machine model (adapted from Del Tedesco et al. (2011)) as shown in Listing 1.1. Assume that before executing the program the buyer's credit card number was read from a terminal and stored in variable `ccNumber`. To complete the purchase, method `buy()` is called and the card account debited. After logging the purchase, the credit card number is erased from memory by setting `ccNumber` to `0`. This ensures that even a powerful attacker who can dump the memory of the vending machine to read the location of variable `ccNumber` cannot learn anything about the credit card number of the buyer.

In more technical terms, we want to ensure that after execution of `buy()` the value of `ccNumber` is permitted to flow to any low location. This policy is not expressible within the standard noninterference framework. A naive and ad hoc extension would be to classify `ccNumber` as high and to add a constraint $f^{\text{ccNumber}}(L, H) \neq 0$ to the insecurity formula. But this does not work, for instance, if we want to erase the secret with a random number. □

To provide support for some of information erasure policies (Del Tedesco et al., 2011; Hunt and Sands, 2008) we generalize our notion of interference (cf. Definition 4).

Definition 6. (Generalized Noninterference). *Given a program* p *over variables Var, a generalized noninterference policy (GNI) is an ordered pair, written as* $H \not\rightsquigarrow_{GNI} L$, *where* $L, H \subseteq Var$ *are sets of low and high variables. Program* p *has secure information flow with respect to GNI if and only if for all concrete executions* $X, Y \in Exc_p$ *it holds that if* $\sigma^X \simeq_{Var \setminus H} \sigma^Y$ *then* $\sigma^X_{out} \simeq_L \sigma^Y_{out}$.

The definition omits the requirement that L and H form a partitioning of Var, i.e. a variable v is allowed to be a member of both variable sets. In addition, it potentially strengthens the condition on the output values of the low variables in the final states. The output values of the low variables in the final states must now be identical for any two initial states that coincide on the variable set of $Var \setminus H$. The set $Var \setminus H$ might not contain all variables in L (e.g., variable v from before would not be in H) and hence allows for more pairs of initial states to be considered. Generalized noninterference reduces to standard noninterference when $L \dot\cup H = Var$.

Example 5. *(Example 4 Continued).* To ensure that the credit card number is erased we specify the *GNI* policy.

$$H = \{\texttt{ccNumber}\} \rightsquigarrow_{GNI} \{\texttt{ccNumber}, \textit{ticketCost}\} = L.$$

To keep the analysis simple, assume that methods charge (**int**, **int**) and `log()` do nothing. First we assume that line 4 that peforms the erasure was forgotten. The following test case demonstrates a violation of the policy: Given initial states σ_1, σ_2 with:

	ccNumber	ticketCost
σ_1	1234	50
σ_2	5678	50

Both initial states coincide on the variable set $Var \setminus H = \{\texttt{ticketCost}\}$. To adhere to the *GNI* policy, the final states $\sigma^1_{out}, \sigma^2_{out}$ must coincide on all low variables, i.e. on the variables `ticketCost` *and* `ccNumber`. But as none of the values is changed by the runs, they still differ in the value of `ccNumber`, hence, the *GNI* property is not valid. Now assume line 4 is present; then, in both executions the final value of `ccNumber` is 0 and the *GNI* property holds.

The logic formalization of the corresponding insecurity formula is almost identical to (2):

$$\bigvee_{l \in L}\bigvee_{0 \le i \le j < n} \overbrace{\left(\left(\bigwedge_{v \in Var \setminus H} v \doteq v'\right) \wedge pc_i(Var) \wedge pc_j(Var') \wedge f^l_i(Var) \ne f^l_j(Var')\right)}^{Leak^{GNI}(H,L,l,i,j)} \quad (3)$$

3.3. *Targeted conditional delimited release*

We further extend the insecurity formula for generalized noninterference (3) to *delimited information release* (DIR) (Sabelfeld and Myers, 2004). In contrast to the standard version of DIR, our policy describes not only *what* information can be released by escape hatches, but also allows to express under which condition and to *whom* (target) the information might be leaked.

Definition 7. (Targeted Conditional Delimited Release). *Given a program* p *over variables Var and a* $GNI = H \not\rightsquigarrow_{GNI} L$. A Targeted Conditional Delimited Release *(TCD) policy* $(\mathcal{D}, GNI)$ *is a set of* specification triples *where each* $(e, C, T) \in \mathcal{D}$ *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 $(\mathcal{D}, GNI)$ *if it satisfies the GNI policy, except for the cases covered by a triple* $(e, C, T) \in \mathcal{D}$. *Here, program is free to release the information captured by the escape hatch expression e to a location in T, provided that condition C is satisfied in the initial state of the execution.*

Given a TCD policy $(\mathcal{D}, GNI)$ 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:

$$\bigvee_{l \in L}\bigvee_{0 \le i \le j < n} (Leak^{GNI}(H, L, l, i, j) \wedge (l \in T \wedge C(Var) \wedge C(Var') \rightarrow e(Var) \doteq e(Var'))) \quad (4)$$

The formula coincides with the noninterference insecurity formula for locations $l \notin T$ that are not among the allowed release targets. Otherwise, the new second conjunct adds:

$$C(Var) \wedge C(Var') \rightarrow e(Var) \doteq e(Var') \quad (5)$$

as an additional restriction to the initial states for both runs: if both initial states satisfy the declassification condition C then they must also coincide on the value of the escape hatch expression. The justification is that if there are two runs such that their initial states coincide 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.

Example 6. *(Example 2 Continued).* Consider the program from Example 2:

```
if (h > 0) {l = 2;}
```

Let $(\{(\texttt{h} > 0, \textit{true}, \{\texttt{l}\}\}, \{\texttt{h}\} \not\rightsquigarrow_{GNI} \{\texttt{l}\})$ be a declassification TCD policy. This is the insecurity formula resulting from symbolic execution starting in (l_0, h_0):

$$(l_0 \doteq l_0' \wedge h_0 > 0 \wedge h_0' > 0 \wedge 2 \ne 2) \wedge ((true \wedge true) \rightarrow (h_0 > 0) \doteq (h_0' > 0)) \vee$$
$$(l_0 \doteq l_0' \wedge h_0 \le 0 \wedge h_0' > 0 \wedge l_0 \ne 2) \wedge ((true \wedge true) \rightarrow (h_0 > 0) \doteq (h_0' > 0)) \vee$$
$$(l_0 \doteq l_0' \wedge h_0 \le 0 \wedge h_0' \le 0 \wedge l_0 \ne l_0') \wedge ((true \wedge true) \rightarrow (h_0 > 0) \doteq (h_0' > 0))$$

The first and third disjuncts are trivially invalid. The second disjunct is invalid, because the second conjunct implies $h_0 > 0$

if and only if $h'_0 > 0$ which contradicts the path condition in the first conjunct. Consequently, the insecurity formula is unsatisfiable, i.e. the program is secure for the specified policy. Consider a slightly altered program:

```
if (h>=0){l=2;}
```

We analyze it with the same policy and initial state as above. Now in the resulting insecurity formula:

$$(l_0 \doteq l'_0 \wedge h_0 \geq 0 \wedge h'_0 \geq 0 \wedge 2 \neq 2) \wedge ((true \wedge true) \rightarrow (h_0 > 0) \doteq (h'_0 > 0)) \vee$$
$$(l_0 \doteq l'_0 \wedge h_0 \leq 0 \wedge h'_0 \geq 0 \wedge l_0 \neq 2) \wedge ((true \wedge true) \rightarrow (h_0 > 0) \doteq (h'_0 > 0)) \vee$$
$$(l_0 \doteq l'_0 \wedge h_0 \leq 0 \wedge h'_0 \leq 0 \wedge l_0 \neq l'_0) \wedge ((true \wedge true) \rightarrow (h_0 > 0) \doteq (h'_0 > 0))$$

the second disjunct is satisfiable, for instance, when the initial value of h is −1 and the initial value of h′ is 0. Consequently, the program does not adhere to the specified policy.

4. Exploit generation using program specifications

Code with unbounded loops or recursive method calls gives rise to infinite symbolic execution trees. Another difficulty is posed by calls to library methods for which no source code is available. And in general symbolic execution trees tend to become infeasibly large when the implementations of called methods are simply inlined. To overcome these problems we annotate programs with specifications in the form of loop invariants and method contracts. This allows to approximate a loop by an invariant and a method call by a contract. In the paper Hentschel et al. (2014) it is shown how to use such specifications during symbolic execution and we can adapt that solution to our setting. Thus it becomes possible to analyze and generate exploits for programs involving unbounded loops or recursive method calls. In our current setting specifications are provided by the user, but in the future we intend to integrate automated inference methods to generate specifications (for example Hähnle et al., 2016; Kovács, 2016; Rodríguez-Carbonell and Kapur, 2007; Wasser, 2015)) into KEG. To keep the presentation readable, in this section we focus on the standard noninterference analysis case. The extension to declassification and erasure is straightforward.

4.1. *Loop specification*

To compute the path conditions and the final values of symbolic execution paths we need to be able to execute unbounded loops without unwinding them infinitely often. In program verification this is achieved 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* that contains at least those program variables the loop can possibly modify.

We need to integrate loop specifications into the NI-insecurity formula (2). Let b be the guard of a loop and $LS = (I, mod)$ its specification. The basic idea in Hentschel et al. (2014) 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 fresh symbolic values V_{mod}. The only knowledge about these values is provided by the loop invariant and by the fact the loop guard b must be *false* after exiting the loop.

Our insecurity formulas express 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 $Leak^{NI}(H, L, l, i, j)$ for the case that both paths i, j contain the same loop:

$$\begin{aligned} Leak^{NI}(H, L, l, i, j) = \Big(\bigwedge_{v \in L} v \doteq v'\Big) \wedge pc_i(V_S) \wedge pc_j(V'_S) \\ \wedge I^{wp}(V_S) \wedge \neg b^{wp}(V_S) \wedge I^{wp}(V'_S) \\ \wedge \neg b^{wp}(V'_S) \wedge f_i^l(V_S) \neq f_j^l(V'_S) \end{aligned} \quad (6)$$

Here $V_S = Var \cup V_{mod}$ and $b^{wp}(V_S)$ are the symbolic values of the guard after the loop expressed in terms of the initial values of V_S. If one or both of paths i, j do not contain the loop or a different loop, then the conjuncts corresponding to the invariants and loop guards are omitted or added accordingly.

Example 7. We illustrate formula (6). Consider the loop below with low variable l and high variable h. We want to establish whether this code is secure with respect to the policy $\{h\} \not\leadsto \{l\}$. The loop specification is $(l \geq 0, \{l\})$. This loop invariant could easily have been inferred with automated methods.

```
l=h*h;

while (l>0){l=l-1;}

l=l+h;
```

Let l_{mod}, l'_{mod} be the fresh values representing the value of l directly after the loop. Computing the weakest precondition of the invariant gives us $l_{mod} \geq 0$ and for the guard $l_{mod} > 0$ for the first run (analogous for the second run). The resulting formula is (note that there is only one path and no path condition):

$$l_0 \doteq l'_0 \wedge l_{mod} \geq 0 \wedge \neg(l_{mod} > 0) \wedge l'_{mod} \geq 0 \wedge \neg(l'_{mod} > 0) \wedge l_{mod} + h_0 \neq l'_{mod} + h'_0$$

The formula is satisfiable, for example, with $l_0 = l'_0 = 10$, $l_{mod} = l'_{mod} = 0$, $h_0 = 1$ and $h'_0 = 2$. Indeed, the program is insecure. Removing the final statement would make it secure. In this case the final conjunct in the insecurity formula would change to $l_{mod} \neq l'_{mod}$ which renders it unsatisfiable. □

4.2. *Method contracts*

Let m be a method name. A contract C_m for m is a triple (Pre_m, $Post_m$, Mod_m) with precondition Pre_m, postcondition $Post_m$ and

modifies (or assignable) clause Mod_m. The latter is the set of all program variables whose value m can possibly change (similar as *mod* in loop specifications).

A method satisfies its contract, if it ensures that when invoked in a state for which the precondition is satisfied, then in the final state the postcondition holds and at most the value of program variables in the assignable clause has been modified.

Analyzing Noninterference Relative to a Precondition. Given a method m with contract C_m. We want to analyze whether m respects a noninterference policy $H \not\rightsquigarrow L$ under the condition that m is only invoked in states satisfying its precondition Pre_m. Adapting the noninterference formula (2) is straightforward and merely requires to add a restriction to the initial states that they must satisfy the method's precondition:

$$Leak^{NI}(H, L, l, i, j) \wedge Pre_m(Var) \wedge Pre_m(Var')$$

Analyzing Programs with Method Contracts for Noninterference. This problem is solved in Hentschel et al. (2014) by using method contracts in a similar way loop specifications have been used. Instead of a loop invariant, the pre- and postconditions become part of the path conditions. The modifies clause again introduces fresh values to represent the symbolic value of program variables that might have been changed as side effect of method invocation.

Let m be the method analyzed for secure information flow and assume it invokes method n. Let the contract of n be (Pre_n, $Post_n$, Mod_n). In the case that each of the paths i, j contains exactly one method call for n we obtain:

$$\begin{aligned} Leak^{NI}(H, L, l, i, j) = \Big(\bigwedge_{v \in L} v \doteq v'\Big) \wedge pc_i(V_S) \wedge pc_j(V_S') \\ \wedge Pre_n^{wp}(V_S) \wedge Post_n^{wp}(V_S) \wedge Pre_n^{wp}(V_S') \\ \wedge Post_n^{wp}(V_S') \wedge f_i^l(V_S) \neq f_j^l(V_S') \end{aligned} \quad (7)$$

where $V_S = Var \cup V_{Mod_n}$ (analagous for the copies) and $Pre_n^{wp}, Post_n^{wp}$ are the weakest preconditions of $Pre_n, Post_n$ computed directly before and after method invocation, respectively. If method n returns a value, a fresh variable representing the return value of n is added to V_S. The return value can be referenced in $Post_n$. The general case (no method call, different method calls, or more than one method call) is handled similarly as in the loop case.

Example 8. We illustrate formula (7) with method run() shown in Listing 1.2. We want to establish whether run() is secure with respect to the policy $\{h\} \not\rightsquigarrow \{l\}$. We expect that it is insecure: the returned value of method calc(**int**) depends on its parameter. In line 3, h is passed as argument, hence, the returned value depends on high input, but it is assigned to low variable l.

We construct the insecurity formula: method run() invokes the recursive method calc(**int**). To analyze the information flow resulting from this invocation, we have used a method contract for calc(**int**), because the recursion does not have a fixed bound.[3] Let calc(**int**)'s contract be given as follows:

[3] Strictly speaking, the Java type **int** and stack size are bounded, but the bound is far too large to be feasible.

Pre_{calc}: true

$Post_{calc}$: $(x \leq 0 \rightarrow \mathtt{result} \doteq 0) \wedge (x > 0 \rightarrow 2 * \mathtt{result} \doteq x * (x+1))$

Mod_{calc}: $\emptyset$

where result refers to the return value and Mod_{calc} is empty as calc(**int**) does not change the state.

Let r be a program variable representing the return value of calc(**int**). To apply the contract for the invocation at line 3, we need to instantiate the above contract as follows:

Pre_{calc}^{wp}: *true*

$Post_{calc}^{wp}$: $(h+2 \leq 0 \rightarrow r \doteq 0) \wedge (h+2 > 0 \rightarrow 2 * r \doteq (h+2) * ((h+2)+1))$

The insecurity formula is then:

$$\begin{aligned} & l_0 = l_0' \wedge (h_0 + 2 \leq 0 \rightarrow r_0 = 0) \wedge (h_0 + 2 > 0 \rightarrow 2r_0 = (h_0 + 2)(h_0 + 3)) \\ & \wedge (h_0' + 2 \leq 0 \rightarrow r_0' = 0) \wedge (h_0' + 2 > 0 \rightarrow 2r_0' = (h_0' + 2)(h_0' + 3)) \\ & \wedge r_0 \neq r_0' \end{aligned}$$

The formula is satisfiable, for example, with $l_0 = l_0' = 10$, $h_0 = 1$, $h_0' = 2$, $r_0 = 6$ and $r_0' = 10$ which means that method run() is insecure. □

Listing 1.2: Recursive method call

```
1  public void run() {
2    h = h + 2;
3    l = calc(h);
4  }
5
6  private int calc (int x) {
7    if (x <= 0) return 0;
8    else        return x + calc(x-1);
9  }
```

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 affect the soundness, but triggers false warnings. The reason is that the specifications might be too weak and allow behaviors 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 hand, if loop or method specifications are not only too weak, but wrong in the sense that they exclude possible behavior, then leaks might go undetected. As we are concerned with bug detection and not verification, this is not too serious as we do not claim to find all bugs. Nevertheless,

incompleteness can be avoided by verifying the specifications using a program verification tool such as KeY (Beckert et al., 2007).

5. Implementation and experiments

5.1. *The KeY exploit generation tool and process*

Our tool *KeY Exploit Generation*[4] (KEG) implements the approach presented in the previous sections. KEG uses the symbolic execution engine of the verification system KeY (Beckert et al., 2007) for the analysis of Java programs. It supports method and loop specifications as explained in Section 4 to compute finite, yet complete (up to the specification) symbolic execution trees. The SMT solver Z3 (de Moura and Bjørner, 2008) is used to find models for the computed insecurity formulas. KEG fully supports Java reference types (objects and arrays). This makes it necessary to test the value of variables that have a reference type for equality. For this we use the approach proposed in Beckert et al. (2014). Comprehension expressions (max, min, Σ) are also supported and can be used in specifications, including escape hatches, loop specifications, and method contracts.

Fig. 2 outlines KEG's workflow. The input is a Java method `m` together with the information flow policy to which it has to adhere. Possibly, `m` is annotated with loop specifications. Contracts of methods called by `m` are provided if they are not to be inlined during symbolic execution. KEG expects the information flow policy to be present in the source code inside specifically marked Java comments. First, method `m` is symbolically executed (using KeY) to obtain a complete symbolic execution tree which can then be queried for the method's path conditions and the final symbolic values of the program locations modified by `m`. Second, using the path conditions and symbolic final values, instances of the insecurity formulas described in Section 3 and 4 are generated. In a third step, these formulas are passed to a model finder (currently we use the SMT solver Z3 (de Moura and Bjørner, 2008)). If a model for the insecurity formula has been found, that model is used to determine the initial states of two runs which exhibit a forbidden information flow. KEG outputs the exploit as a self-contained JUnit test, which can then be included into a regression test suite. The generated exploit executes two runs (one for each initial state) and inspects the final states to detect a leak in form of an assertion. In this way the conjuncts in the insecurity formulas that contain the inequality over symbolic final values can be viewed as an automatically synthesized test oracle.

Listing 1.3: Example program to illustrate KEG

```
1 public class Simple {
2   public int l;
3   private int x, y;
4   /*! l | x y ; !*/
5
6   /*@ escapes (x*y) \to l \if x>-1; @*/
7   public void magic() { if (x>0) { l=x*y; } else { l=0; } }
8 }
```

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

5.2. *Exploit generation by example*

We illustrate the work flow of KEG step by step with the program in Listing 1.3. Class `Simple` declares three integer typed fields `l`, `x`, `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 of the class is $\{x, y\} \not\rightarrow \{l\}$ and specified in a comment starting with "/*!" in Line 4. Variables `x`, `y` are implicitly declared as high variables and `l` as a low variable. This strict noninterference policy is relaxed in line 6 for method `magic()` by providing a targeted 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 unique pair of these paths the corresponding insecurity formulas and passes these on SMT solver. Only one of the three generated insecurity formulas is satisfiable by the following model provided by Z3:

Insecurity formula (in SMT-LIB syntax)	Model

`(let ((a!1 (not (and (>x_1 (- 1))`
`(>x_2 (- 1))))))`
`(and (>= x_1 1) (<= x_2 0)`
`(or (not (=x_1 x_2)) (not (=y_1 y_2)))`
`(=l_1 l_2) (not (=(* y_1 x_1) 0))`
`(or a!1 (=(* x_1 y_1) (* x_2 y_2)))))`
	`x_1: 1`
	`x_2: -1`
	`y_1: 1`
	`y_2: -1`
	`l_1: 0`
	`l_2: 0`

The model describes two runs: The first run, labeled with 1, starts in an initial state with `x`, `y` and `l` initialized with 1, 1 and 0, respectively. We identify variable `v` in execution X by `v_X`. The second run, labeled with 2, has initial values −1, −1, and 0. The first run enters the then-branch of the conditional, the second one does not.

As the value of `l` is altered in the first run when executing the then-branch but not by the second run, where it remains 0, a leak is detected (the leaked information is the sign of field `x`). KEG generates exactly one exploit, which is output as a well-structured and human readable JUnit test. The exploit program is depicted in Listing 1.4. KEG outputs the exploit exactly as shown, i.e. pretty printed and structured with comments. We have only renamed a few fields to increase readability further.

The two initial states for the two runs of method `magic` are set up in lines 6–14 and 23–31. To ensure that the runs do not interfere, two instances of class `Simple` are created (lines 6 and 23). In general, more work needs to be invested to ensure independent runs (for instance, if static members are modified by a run). This can be achieved by running the program on two different Java Virtual Machines and by querying those for the required information. (This is currently not supported by KEG.)

Before invoking method `magic`, the initial values of all fields and method parameters are assigned. In Listing 1.4, the initial values of each field `l`, `x`, `y` of both runs are stored in the corresponding variables `l_1`, `x_1`, `y_1` (lines 7–9) and `l_2`, `x_2`, `y_2` (lines 24–26). These initial values are taken from the counter example produced by the SMT solver. To assign values to the

Table 1 – Benchmark 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
ArrSearch	search	1/0	D	S	4199	2934	2400	9568	0/0
ArrMax	findMax	1/0	NI	I	4215	3584	963	8804	1/0
ArrMin	findMin	1/0	D	S	4746	3128	983	8925	0/0
ArrSum	calcSum	1/0	D	S	5481	2504	788	8846	0/0

#(L/MI/GE/FW): nr of loops/method invocations/generated exploits/false warnings.
NI/D: Non-interference/declassification, S/I: secure/insecure.
T_X: time for loading/symbolic execution/model finding/total.

Listing 1.4: JUnit test as exploit program

```
1  @Test
2  public void test_magic_l_0 ()
3  throws NoSuchFieldException, SecurityException,
4         IllegalArgumentException, IllegalAccessException {
5    /* Prepare for execution 1 */
6    Simple s1 = new Simple();
7    int l_1 = 0;
8    int x_1 = 1;
9    int y_1 = 1;
10
11   /* Configure variable: s1 */
12   setFieldValue(s1,"l",l_1);
13   setFieldValue(s1,"x",x_1);
14   setFieldValue(s1,"y",y_1);
15
16   /* Perform execution 1 */
17   s1.magic();
18
19   /* Get the value of low variable l after execution 1 */
20   int l_out_1 = ((Integer)getFieldValue(s1,"l")).intValue();
21
22   /* Prepare for execution 2 */
23   Simple s2 = new Simple();
24   int l_2 = 0;
25   int x_2 = -1;
26   int y_2 = -1;
27
28   /* Configure variable: s2 */
29   setFieldValue(s2,"l",l_2);
30   setFieldValue(s2,"x",x_2);
31   setFieldValue(s2,"y",y_2);
32
33   /* Perform execution 2 */
34   s2.magic();
35
36   /* Get the value of low variable l after execution 2 */
37   int l_out_2 = ((Integer)getFieldValue(s2,"l")).intValue();
38
39   assertNotNull(l_out_1);
40   assertNotNull(l_out_2);
41   /*
42   * assert that the value of low variable l is not changed after
       performing
43   * two executions
44   */
45   assertTrue(l_out_1 == l_out_2);
46 }
```

fields of an object, the auxiliary method `setFieldValue` makes use of Java's reflection framework.

After each run (at line 17 and line 34), the concrete output value of `l` is extracted (lines 20 and 37). Line 45 asserts that the output values of `l` observed at the end of each run are equal. The JUnit test throws an assertion failure exception, if the output values of the two runs are different and thus an actual leak happened. If no exception is thrown then the counter example was spurious, for instance, due to a too weak loop specification. Tests that do not result in an assertion failure can then be omitted from our test suite.

5.3. *Experiments*

We performed experiments[5] on micro benchmarks as a first part of the 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, for example, by using a hand-coded version instead of a generated parser. Model finding time can be further 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 that is run only few times will not be optimized at all.

A few observations concerning the benchmarks: For the examples *Mul* and *Comp*, we analyzed the effect of loop and method specifications in case of strong, weak and wrong specifications (*filename*_Strong/Weak/Wrong_LI/MC). As expected, with sufficiently strong specifications, all insecure paths could be precisely identified and only actual exploits were generated. Weak specifications over-approximate the behavior, leading to false positives, while wrong specifications can prevent to analyze all possible

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

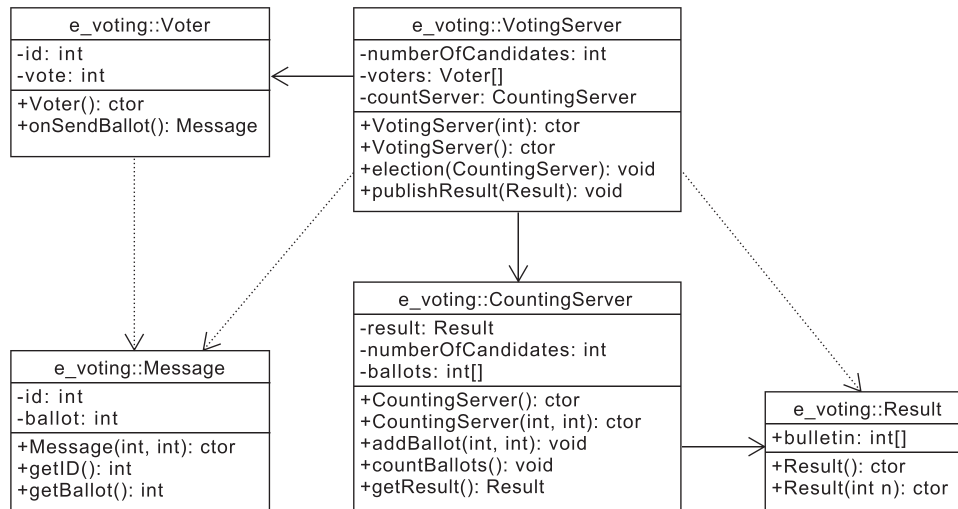


Fig. 3 – UML class diagram of the e-voting system in the case study.

behaviors and some existing leaks were missed. The analysis of method `search` in classes *ArrSearch*, *ArrMin*, *ArrSum* identified the method correctly as secure with respect to the specified declassification policy and generated no exploits.

6. E-voting case study

For our case-study we adapted the electronic voting system presented in Grahl (2015) and Scheben (2014), which is based on sElect, a real-world electronic voting system (Küsters et al., 2011) implemented in Java. Electronic voting systems rise a plethora of security issues like *integrity*, *verifiability* and *coercion-resistance*. We focus on the *confidentiality* of individual ballots, i.e. we aim to detect all possible information flow leaks from individual ballots to the public result.

6.1. *Simplified E-voting system*

Vote confidentiality in sElect is guaranteed by the use of cryptography. Cryptographic algorithms and protocols are based on advanced mathematical theories (and even unproven assumptions about the hardness of an underlying mathematical problem). This makes it infeasible to verify sElect using current information flow analysis tools.

To be able to analyze such systems, Küsters et al. (2011) proposed a solution using *ideal encryption*. In a nutshell, their solution removes the encryption component from the system and enables the use of information flow analysis tools, which can now be run on the “simpler” program. The authors prove that information flow analysis results for the transformed program keep their validity for the original system.

Grahl (2015) and Scheben (2014) apply the formal verification tool KeY on an e-voting system that contains the essential components of sElect. They focus on formal verification of vote confidentiality and integrity. Even though their e-voting program is not distributed and does not contain complex features, i.e. cryptography and networking, its verification requires considerable effort and user interaction.

Fig. 3 shows a UML class diagram of the e-voting system used in our case study[6], which is based on the implementation presented in Grahl (2015). We redesigned, but did not simplify, the system slightly to be able to show the capabilities of KEG, in particular, its support for information erasure policies. It consists of five classes: `VotingServer`, `CountingServer`, `Voter`, `Message` and `Result`. The voting protocol is as follows: First, voters (class `Voter`) register and obtain a unique identifier from the voting server (class `VotingServer`). Then, they send their vote to the server using a message (class `Message`) composed of the voter’s identifier and vote. Voters are not allowed to change their vote once cast, even if the voting is still ongoing. The voting server receives the messages sent by the voters and forwards the ballots to the counting server (class `CountingServer`). Once all voters have cast their vote, the counting server computes the election result and returns it to the voting server. The result is then published. The counting server must not keep any ballots after the election result has been computed.

Listing 1.5: Class `VotingServer`

```
1  public class VotingServer {
2    private int numberOfCandidates;
3    private Voter[] voters;
4    private CountingServer countServer;
5    ...
6    public void election(){
7      for (int i=0; i<voters.length; i++){
8        Message message = voters[i].onSendBallot();
9        countServer.addBallot(i, message.getBallot());
10     }
11     countServer.countBallots();
12     publishResult(countServer.getResult());
13   }
14 }
```

Class `VotingServer` (Listing 1.5) is responsible for the overall election process which is coordinated by method `election()`.

[6] www.se.tu-darmstadt.de/research/projects/albia/download/e-voting-declassification-erasure.

Each voter is queried for her/his ballot, which is then passed on to the counting server (lines 7–10). After all voters cast their vote, the counting server starts counting all ballots (line 11) and computes the election's result. Finally, the voting server publishes the computed election result (line 12).

Listing 1.6: Class CountingServer

```
1  public class CountingServer {
2    private Result result;
3    private int numberOfCandidates;
4    private int[] ballots;
5    /*! result ballots numberOfCandidates | ballots ; !*/
6    ...
7    public void addBallot(int idx, int vote) { ballots[idx] = vote; }
8
9    /*@ requires numberOfCandidates>0 && ballots.length>0; @*/
10   public void countBallots() {
11     result = new Result(numberOfCandidates);
12     for (int i=0; i<ballots.length; i++)
13     if (ballots[i] >= 0 && ballots[i] < numberOfCandidates) {
14       result.bulletin[ballots[i]]++;
15     }
16   }
17
18   public Result getResult() { return result; }
19 }
```

The relevant methods of class `CountingServer` are shown in Listing 1.6. Its field `result` keeps the bulletin, which is the aggregated result of all ballots counted so far. Counting must only take place, if there is at least one candidate and one ballot. This assumption is specified in the precondition of method `countBallots()` at line 9. Line 5 specifies the generalized noninterference policy:

$$\{\texttt{ballots}\} \not\rightsquigarrow_{GNI} \{\texttt{result}, \texttt{ballots}, \texttt{numberOfCandidates}\}$$

which disallows any information flow from `ballots` to any other field, including `ballots` itself. If an object or array appears in a noninterference policy, then KEG also includes in the policy all of its fields or elements, respectively. The election protocol described above requires the deletion of all ballots once `countBallots()` finished computing the result. Hence, the *GNI* policy contains the field `ballots` on both sides to enforce the field's erasure as well as the erasure of all elements of the referred array.

To be able to analyze method `countBallots()` with respect to secure information flow, KEG needs to reason about the flow of information within the (unbounded) loop iterating over all ballots. Hence, a correct and sufficiently strong loop specification is required. Listing 1.7 shows one possible loop invariant which guarantees that all ballots are counted correctly. It is quite simple and expresses (a) that the loop counter i stays within valid bounds and (b) that all ballots up to i have been correctly counted. The assignable clause (line 6) states that the loop may modify all elements of the array `result.bulletin` and the loop counter.

Listing 1.7: Loop invariant in method countBallots

```
1 /*@ loop_invariant
2   @   i>=0 && i<=ballots.length &&
3   @     (\forall int k; k >= 0 && k < numberOfCandidates;
4   @         result.bulletin[k] == (\sum int j; 0 <= j && j < i;
5   @                                    (k==ballots[j]?1:0)));
6   @ assignable result.bulletin[*], i;
7   @*/
```

6.2. Checking noninterference and declassification

To analyze class `CountingServer` with respect to the *GNI* policy defined above we ran KEG on a Macbook Pro Retina late 2013 (2.6 GHz Intel Core i5 processor, 8 GiB RAM, Mac OS X 10.11.5). By default, KEG analyzes all public methods of a class. After 63 seconds KEG finished its analysis and generated seven exploits showcasing different violations of the specified *GNI*. Listing 1.8 shows one of the generated exploits exposing an information leak from `ballots` to `result` in method `countBallot()`.

Running the generated exploits results in assertion failures for all of them which means that genuine leaks were found. Looking closer at the generated exploit in Listing 1.8, we see that both initial states (lines 5–13 and lines 18–25) are identical except for the content of the `ballots` array whose entries are set to 2006 in the first initial state and to 0 in the second one. Consequently, the assertion failure in line 31 must be the result of an information flow from `ballots` (or its contents) to the aggregated result `result.bulletin`.

This is not very surprising, because the outcome of an election depends on the votes. We have to fix our policy by allowing some information to be leaked to the election result, namely, the aggregated number of votes per candidate. To specify this we use the TCD policy introduced in Section 3.3 and relax the noninterference policy of method `countBallots()` by adding an escape hatch expression:

```
@ escapes
@    (\seq_def int i; 0; numberOfCandidates;
@       (\num_of int j;
@            0 <=j && j < ballots.length; i ==ballots[j]));
```

Listing 1.8: Exploit program for class CountingServer

```
1  public class TestCountingServer extends TestCase {
2    @Test
3    public void test_countBallots_result_bulletin_0()
4    throws NoSuchFieldException, ... {
5    /* Prepare for execution 1 */
6    CountingServer cs_1 = new CountingServer();
7    e_voting.Result cs_result_1 = new e_voting.Result();
8    ...
9    int cs_numberOfCandidates_1 = 2006;
10   int[] cs_ballots_1 = new int[1];
11   ...
12   /* Configure variable: cs_ballots_1 */
13   for (int i=0; i<cs_ballots_1.length; i++) cs_ballots_1[i] = 2006;
14   /* Perform execution 1 */
15   cs_1.countBallots();
16
17   /* Prepare for execution 2 */
18   CountingServer cs_2 = new CountingServer();
19   e_voting.Result cs_result_2 = new e_voting.Result();
20   ...
21   int cs_numberOfCandidates_2 = 2006;
22   int[] cs_ballots_2 = new int[1];
23   ...
24   /* Configure variable: cs_ballots_2 */
25   for (int i=0; i<cs_ballots_2.length; i++) cs_ballots_2[i] = 0;
26   /* Perform execution 2 */
27   cs_2.countBallots();
28   ...
29   /* assert that the value of low variable cs_result_bulletin
30    * is not changed after performing two executions */
31   Assert.assertArrayEquals(cs_result_bulletin_out_1,
32                            cs_result_bulletin_out_2);
33   }
34 }
```

The escape hatch expression uses JML's **\seq_def** constructor to define a sequence whose i-th element is equal to the number of votes cast for the i-th candidate. This means that the program is allowed to leak the number of votes for each candidate. For each candidate i, the comprehension expression **\num_of** returns the number j of indices between 0 (inclusive) and `ballots.length` (exclusive) that satisfy the Boolean expression `i = =ballots[j]`.

After changing the policy we rerun KEG on `CountingServer`. It finds six exploits in ca. 50 seconds, one less than before. Running the exploits results again in assertion failures for all which means they are genuine. But inspection of the generated exploits shows that none of them indicates an information flow from votes to the result. This is a strong hint that no such leak exists (by proving the loop invariant and method contracts this can even be verified). Therefore, the problem must lie elsewhere. By similar reasoning as above, it can be easily seen that the test cases fail, because the information erasure policy `ballots` $\not\rightarrow_{GNI}$ `ballots` is violated by methods `countBallot()`, `addBallot(int, int)` and `getResult()`. Inspecting the source code reveals that the ballots are not erased at all.

Let us now fix these issues one method at a time. We decide that method `countBallot()` is responsible to erase the individual ballots once it computed the result. To do this, we add the new private method `clearBallots()` to class `CountingServer`

```
private void clearBallots() { ballots = new int[0]; }
```

and add an invocation of `clearBallots()` as a final statement to `countBallots()`. Now those exploits related to `countBallots()` pass without assertion failure.

We turn to method `getResult()`. The exploits related to it still fail, because the method does not erase any information about the ballots. If the method were only called after `countBallots()` terminates, this would actually be fine, because `countBallots()` erases all information. This shows that one has to be extremely careful when refactoring security-critical code: seemingly harmless rearrangements can introduce subtle leaks.

One can argue in favor of defensive programming and simply add a call to `clearBallots()` to `getResult()`. Another strategy is to weaken the security policy. We discuss this possibility now for method `addBallot(int, int)`.

Assume we fixed `getResult()`, then the remaining failing test cases are related to method `addBallot(int, int)`. This method is used to collect the individual votes before the result is computed. The solution to erase the ballots is not applicable, because method `countBallots()` needs this information. Instead, we decide to alter the information flow policy for this method by adding the escape hatch expression

```
@ escapes ballots;
```

as a local method annotation. This "deactivates" the information erasure requirement, but still enforces the noninterference part of the policy. We run KEG now on the corrected version of `CountingServer`. KEG finishes without generating any exploits after 20 seconds.

6.3. *Discussion*

In contrast to Grahl (2015) and Scheben (2014) our main interest is not to formally verify that the given program is secure, but to detect *and* to demonstrate the existence of leaks. The case study shows that information erasure can be represented as a generalized noninterference policy and be actually checked by KEG in practice.

We showed that KEG can be applied to object-oriented program with unbounded loops. KEG was able to generate exploits that demonstrated violations of the specified information flow policy. The exploits assisted in identifying and fixing the existing leaks. The fixes were validated by checking that the generated exploits passed and that KEG was not able to generate any new ones. Except for the provision of the specifications, the approach does not require any user interaction. Specifically, no expert knowledge in logic or theorem proving is required.

KEG can be integrated with specification generation techniques (Kovács, 2016; Rodríguez-Carbonell and Kapur, 2007; Wasser, 2015) to reduce the need of user-specified loop invariants and method contracts. In Do et al. (2016) we successfully used the abstraction framework of Hähnle et al. (2016) to automatically generate suitable specifications for information flow analysis.

Some words on scalability for real-world programs. Our approach is contract-based and thus only one (or very few methods) needs to be considered at one time. It is also possible to restrict the analysis to critical modules and thus to reduce the number of required additional specifications like loop invariants. In addition, the approach is more about bug finding than verification, so even simple contracts and loop invariants are useful.

7. Related work

Our approach to exploit generation is based on self-composition (Barthe et al., 2004; Darvas et al., 2003, 2005). The paper Darvas et al. (2005) 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 follows techniques that were first explored in automatic test generation. In particular, we build on work presented in Albert et al. (2010), de Halleux and Tillmann (2008) Engel and Hähnle (2007), and King (1976), where symbolic execution is used as a means to generate test cases for functional properties.

Deductive approaches to information flow analysis (Beckert et al., 2014; Scheben and Schmitt, 2012) 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 nontrivial interactions with a theorem prover are required. This restricts usability of these approaches seriously. In Nanevski et al. (2011) higher-order logic is used to express information flow properties for object-oriented programs, which is highly expressive, but imposes even higher demands on user expertise.

Pairs of symbolic execution paths to improve the efficiency of self-composition have been independently introduced in Phan (2013) to check programs for noninterference. However, that paper focusses on checking noninterference and does not

support declassification. Unbounded loops and recursive methods are not handled either.

In Vaughan and Chong (2011), leaks are inferred automatically and expressed in a human-readable security policy language, helping programmers to decide whether the program is secure, however, they cannot give concrete counter examples that could suggest further corrections. Counter examples can be used not only to generate executable exploits as in our approach, but also to refine declassification policies by quantifying the leakage (Backes et al., 2009; Banerjee et al., 2007). However, none of these approaches provides a solution for unbounded loops and recursion.

ENCoVer (Balliu et al., 2012) uses epistemic logic and makes use of symbolic execution (concolic testing) to check noninterference for Java programs. In Milushev et al. (2012), the authors proposed a tool which checks that a C program is secure with respect to 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 (Hunt and Sands, 2006; Myers, 1999; Sabelfeld and Myers, 2004; Volpano et al., 1996) and those based on dependency graphs (Graf et al., 2013) distinguish themselves by their high performance and ability to check large systems. Their common drawbacks are a lack of precision with a resulting high number of false positives and restrictions on the syntactic form of programs.

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 these approaches, but is intended to be used complementary, just like testing complements formal verification.

In Do et al. (2016) KEG was combined with a specification generation tool to reduce the necessity of providing loop invariants or method specifications manually. The focus was on noninterference of a privacy game derived from sElect (Küsters et al., 2011, 2015). The privacy game is much simpler than the e-voting system in Section 6. In fact, using a privacy game was in part motivated by the need to avoid declassification.

8. Conclusion

We presented a novel approach for automatic detection of information flow leaks in object-oriented programs. Exploits are generated based on satisfying models of *insecurity formulas* and output as self-contained JUnit tests so that they can easily be integrated into regression test libraries. We demonstrated how program specifications such as loop invariants and method contracts can be used to deal with unbounded program structures that otherwise give rise to infinite symbolic execution trees. We built a fully automatic tool (KEG) based on our approach that handles sequential Java programs and we applied it to a number of case studies, including an executable e-voting program.

REFERENCES

Albert E, Gomez-Zamalloa M, Puebla G. PET: a partial evaluation-based test case generation tool for Java bytecode. In: ACM SIGPLAN workshop on partial evaluation and semantics-based program manipulation (PEPM). pp. 25–28. ACM Press; 2010.

Askarov A, Chong S, Mantel H. Hybrid monitors for concurrent noninterference. In: Fournet, C., Hicks, M.W., Viganò, L. (eds.) IEEE 28th computer security foundations symp., CSF, Verona, Italy. pp. 137–151. IEEE Computer Society; 2015.

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 '09, IEEE CS; 2009.

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.

Banerjee A, Naumann DA. Stack-based access control and secure information flow. J Funct Program 2005;15(2):131–77.

Banerjee A, Giacobazzi R, Mastroeni I. What you lose is what you leak: information leakage in declassification policies. Electr Notes Theoretic Comput Sci 2007;173:47–66.

Barr ET, Harman M, McMinn P, Shahbaz M, Yoo S. The oracle problem in software testing: a survey. IEEE Trans Softw Eng 2015;41(5):507–25.

Barthe G, D'Argenio PR, Rezk T. Secure information flow by self-composition. In: Proc. of the 17th IEEE workshop on computer security foundations. pp. 100–114. CSFW '04, IEEE CS; 2004.

Barthe G, Crespo JM, Kunz C. Relational Verification Using Product Programs. In: Proc. of the 17th intl. conf. on formal methods. pp. 200–214. FM'11, Springer; 2011.

Beckert B, Hähnle R. Reasoning and verification. IEEE Intell Syst 2014;29(1):20–9.

Beckert B, Hähnle R, Schmitt PH. Verification of Object-oriented Software: The KeY Approach. Springer; 2007.

Beckert B, Bruns D, Klebanov V, Scheben C, Schmitt PH, Ulbrich M. Information Flow in Object-Oriented Software. In: Gupta, G. (ed.) Post proc. logic-based program synthesis and transformation. LNCS, Springer; 2014.

Cohen ES. Information Transmission in Sequential Programs. Foundations of secure computation pp. 297–335; 1978.

de Halleux J, Tillmann N. Parameterized unit testing with Pex. In: Beckert, B., Hähnle, R. (eds.) Tests and proofs, second international conference, TAP 2008, Prato, Italy, April 9–11, 2008. Proceedings. LNCS, vol. 4966, pp. 171–181. Springer; 2008.

de Moura L, Bjørner N. Z3: An Efficient SMT Solver. In: Ramakrishnan, C., Rehof, J. (eds.) Tools and algorithms for the construction and analysis of systems, LNCS, vol. 4963, pp. 337–340. Springer; 2008.

Dam M, Jacobs B, Lundblad A, Piessens F. Provably correct inline monitoring for multithreaded Java-like programs. J Comput Secur 2010;18(1):37–59.

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.

Darvas A, Hähnle R, Sands D. A theorem proving approach to analysis of secure information flow. In: Proc. of the 2nd intl. conf. on security in pervasive computing. pp. 193–209. SPC'05, Springer; 2005.

Del Tedesco F, Hunt S, Sands D. A semantic hierarchy for erasure policies. In: Jajodia, S., Mazumdar, C. (eds.) Information systems security: 7th intl. conf., ICISS, Kolkata, India. pp. 352–369. Springer; 2011.

Devriese D, Piessens F. Noninterference through secure multi-execution. In: 31st IEEE symp. on security and privacy, S&P, Berleley/Oakland, USA. pp. 109–124. IEEE Computer Society; 2010.

Do QH, Bubel R, Hähnle R. Exploit generation for information flow leaks in object-oriented programs. In: Federath, H., Gollmann, D. (eds.) Proc. 30th Intl. IFIP conf. on ICT systems

security and privacy protection. IFIP advances in information and communication technology, vol. 455, pp. 401–415. Springer; 2015.
Do QH, Kamburjan E, Wasser N. Towards fully automatic logic-based information flow analysis: An electronic-voting case study. In: Piessens, F., Viganò, L. (eds.) Principles of security and trust, 5th intl. conf., POST, Eindhoven, The Netherlands. LNCS, vol. 9635, pp. 97–115. Springer; 2016.
Engel C, Hähnle R. Generating unit tests from formal proofs. In: Meyer, B., Gurevich, Y. (eds.) Proc. of tests and proofs. LNCS, vol. 4454. Springer; 2007.
Gladisch C. Verification-based test case generation for full feasible branch coverage. In: Cerone, A., Gruner, S. (eds.) Proc. 6th IEEE intl. conf. on software engineering and formal methods, SEFM, Cape Town, South Africa. pp. 159–168. IEEE Computer Society; 2008.
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; 2013.
Grahl D. Deductive verification of concurrent programs and its application to secure information flow for Java [Ph.D. thesis]. Karlsruhe Institute of Technology; 2015. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000050695.
Hähnle R, Wasser N, Bubel R. Array abstraction with symbolic pivots. In: Ábrahám, E., Bonsangue, M., Johnsen, E.B. (eds.) Theory and practice of formal methods: essays dedicated to Frank de Boer on the occasion of His 60th birthday, LNCS, vol. 9660, pp. 104–121. Springer; 2016.
Hentschel M, Hähnle R, Bubel R. Visualizing unbounded symbolic execution. In: Proc. of tests and proofs, pp. 82–98. Springer; 2014.
Hunt S, Sands D. On flow-sensitive security types. In: ACM SIGPLAN notices. vol. 41, pp. 79–90. ACM; 2006.
Hunt S, Sands D. Just forget it – the semantics and enforcement of information erasure. In: Programming languages and systems. 17th European symposium on programming, ESOP 2008. pp. 239–253. No. 4960 in LNCS, Springer Verlag; 2008.
King JC. Symbolic execution and program testing. Commun ACM 1976;19(7):385–94.
Kovács L. Symbolic computation and automated reasoning for program analysis. In: Ábrahám, E., Huisman, M. (eds.) Integrated formal methods, 12th intl. conf., IFM, Reykjavik, Iceland. LNCS, vol. 9681, pp. 20–27. Springer; 2016.
Küsters R, Truderung T, Vogt A. Verifiability, privacy, and coercion-resistance: New insights from a case study. In: 32nd IEEE symp. on security and privacy, S&P, Berkeley, CA, USA. pp. 538–553; 2011.
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.
Milushev D, Beck W, Clarke D. Noninterference via symbolic execution. In: Proc. of FMOODS'12/FORTE'12. pp. 152–168. Springer; 2012.
Myers AC. JFlow: Practical Mostly-Static Information Flow Control. In: Proc. of 26th ACM symp. on principles of programming languages. pp. 228–241; 1999.
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 '11, IEEE CS; 2011.
Phan QS. 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.
Rodríguez-Carbonell E, Kapur D. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. Sci Comput Program 2007;64(1): 54–75.
Sabelfeld A, Myers AC. A model for delimited information release. In: Software security-theories and systems, pp. 174–191. Springer; 2004.
Sabelfeld A, Sands D. Declassification: dimensions and principles. J Comput Secur 2009;17(5):517–48.
Scheben C. Program-level specification and deductive verification of security properties [Ph.D. thesis]. Karlsruhe Institute of Technology; 2014. http://digbib.ubka.uni-karlsruhe.de/volltexte/1000046878.
Scheben C, Schmitt PH. Verification of information flow properties of Java programs without approximations. In: Formal verification of object-oriented software, LNCS, vol. 7421, pp. 232–249. Springer; 2012.
Terauchi T, Aiken A. Secure information flow as a safety problem. In: Proc. of the 12th intl. conf. static analysis (SAS). pp. 352–367. LNCS, Springer; 2005.
Vaughan JA, Chong S. Inference of expressive declassification policies. In: Proc. of the 2011 IEEE symp. on security and privacy. pp. 180–195. IEEE CS; 2011.
Volpano D, Irvine C, Smith G. A sound type system for secure flow analysis. J Comput Secur 1996;4(2):167–87.
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, first intl. symp., Nanjing, China. LNCS, vol. 9409, pp. 243–257. Springer; 2015.

Quoc Huy Do received his masters degree in Computer Science in 2010 at the Vietnam National University, Hanoi. Currently he is a Ph.D. student at the Computer Science Department of the Technical University of Darmstadt. He participated in the DFG funded project ALBIA (part of the DFG Priority programme RS3). His main interests are formal methods in software development with a focus on information flow analysis.

Richard Bubel graduated in 2003 at the University of Karlsruhe (Germany) in Computer Science and obtained his Ph.D. there in 2007. He worked as a postdoc at the Chalmers University in Gothenburg. Currently he is employed as postdoc at the Computer Science Department of the Technical University of Darmstadt. He participated in the EU projects MOBIUS, HATS and Envisage. He was a principal investigator in the DFG funded project ALBIA (part of the DFG Priority programme RS3). His main interests are deductive program verification, theorem proving and formal methods in software development.

Reiner Hähnle holds the Chair of Software Engineering at the Computer Science Department of Technische Universität Darmstadt. He received his Ph.D. in Computer Science from University of Karlsruhe in 1992. From 2000 to 2011 he worked as an Associate, later as Full Professor at Chalmers University of Technology, Gothenburg. Since 2011 he is with TU Darmstadt. His main research interest is to apply Formal Methods in Software Engineering and in Computer Security.