KEG:
The KeY Exploit Generation tool
User Guide

Huy Q. Do, Richard Bubel, Reiner Hähnle
email: do@cs.tu-darmstadt.de

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Software Engineering group
Department of Computer Science
TU Darmstadt

Abstract

This tutorial supplies necessary guidelines on using KEG to check Java program and find useful exploits w.r.t. information flow security.

1 Introduction

KEG (KeY Exploit Generation) is a tool detecting leakages in Java programs and generating exploits that could help developers identify vulnerabilities, locate and revise risky source code as well as test their programs with respect to information flow security polices.

This tutorial describes step by step how Java programmers can use KEG to check their programs. It contains following sections:

- 2 JML specification: Introduces about JML specifications that are necessary for running KEG
- 3 Specification for Information Flow Security Policies: Describes the syntax specifying for information flow security policies, including noninterference and declassification
- 4 Running KEG: Introduces about how to run KEG to check specified Java programs

2 JML specification

KEG uses KeY [1] as the back-end to symbolically analyse Java programs based on proof obligations. Proof obligations are generated via method contracts (or functional operation contract). To check a method of a Java class, it is necessary to specify it's method contract in term of JML specification [2]. A method contract normally consists of three parts: precondition, postcondition and modifies clause. To check a method by KEG w.r.t. information flow security, at least precondition or postcondition must be specified.

KEG solves a method call inside checking method in two ways: inlining the invoked method or using it's own method contract. The first solution is only feasible to simple, bounded methods. If the invoked method itself contains unbounded loop recursion structures, the second solution must be used. In this case, we have to define full method contract specification for invoked method, including precondition, postcondition and assignable clause. For more detail about method contract specification in KeY, please read [1].

If checking method containing itself loop structures, KEG also supplies two solutions: unwinding the loop or using loop invariant. The first solutions is only feasible for bounded loop, whereas the second must be used for resolving unbounded loop structure. For more detail about specifying loop invariant, please read [1].

Example 2.1. Listing 1 shows an example of method contract and loop invariant specification. Method contract of `magic` is specified from line 5 to line line 9. Line 6 defines precondition, line 7 is postcondition and line 8 is assignable clause specifying that the value of `l` is modified along with the execution of method `magic`. Lines 14 - 18 define specification for the next while-loop, which line 15 is loop invariant, line 16 is assignable clause claiming that the value of `m` and `i` will be changed inside the while-loop structure.

3 Specification for Information Flow Security Policies

3.1 Noninterference

Each noninterference policy, treated by KEG as a class-level one, consists of two elements: source (high variables) and sink (low variables). It prohibits all information flow from source to sink. To define a noninterference policy, you have to point out its corresponding source and sink. The syntax specifying a noninterference policy is as followings:

```
/*! sink | source ; !*/
```

Where *sink* and *source* is two sets of memory locations in the program. They could be not only class's fields, but also method's parameters and array's elements. At the moment, KEG only supports primitive type (integer or boolean) w.r.t. memory locations in source and sink parts.

If there are more than one noninterference policy, they can be defined using following syntax:

```
/*!
  sink_1 | source_1 ;
  sink_2 | source_2 ;
  ...
  sink_n | source_n ;
!*/
```

Listing 1: JML specification: method contract and loop invariant

```
1  public class Loop {
2    public int l;
3    private int x, y;
4
5    /*@
6    @requires x>0;
7    @ensures l==x*y;
8    @assignable l;
9    @*/
10   public void magic(){
11     if (x>0){
12       int m=0;
13       int i=0;
14       /*@
15       @ loop_invariant (i<=x)&&(m==i*y);
16       @ assignable m,i;
17       @ decreases x-i;
18       @*/
19       while (i < x) {
20         m+=y;
21         i++;
22       }
23       l = m;
24     } else l = 0;
25   }
26 }
```

Example 3.1. Let us look into class `Accountant` at listing 2. It has five fields and one method. Noninterference policies are defined from line 5 to line 8, which line 5 and line 8 are opening and closing marks for noninterference specification whereas line 6 and line 7 are two noninterference policies claiming that `sum` and `flag` cannot interfere salaries of `a` and `b`; whereas `flag` cannot interfere `password`. These policies are class-level which means that they are applied for all method of class `Accountant`. Line 9 specifies precondition of `magic` that is assumed to be satisfied before `magic` is invoked.

3.2 Declassification

Noninterference is the strongest policy for information flow security. It is usually too strict for practical applications. To relax it, declassification policy which allows to leak some information from secret locations can be used. For instance, back to example 3.1, we might want to allow `sum` store the total salary of `a` and `b` w.r.t. the execution of method 'codemagic. KEG supports conditional delimited release for declassification and treats it as a method-level policy which its specification is embedded in method contract specification. The syntax of conditional delimited release policy is as following:

```
@escapes  released expression E  [\if condition C ] [\to destinations D ];
```

Above syntax defines following declassification policy: secret information can be leaked through expression`E` to destinations `D` (set of memory locations) if condition `C` is satisfied before `magic` is executed. To use method calls within expressions `E` and `C`, we need to supply them corresponding method contracts.

Example 3.2. Line 8 of listing 3 yields the policy that allows the total salary of `a` and `b` to leak to `sum` if `pass > -1` is satisfied before `magic` is executed. This policy is re-expressed in listing 4, which escapes hatch expression `(a.salary + b.salary)` is replaced by method call `sum(a.salary, b.salary)`. The semantics of method `sum` is supplied by its method contract at line 14.

4 Running KEG

KEG currently only supports command line interface. The command to run KEG is:

```
java -jar ⟨⟨ path to exploitgen.jar ⟩⟩ ⟨⟨ Java source file ⟩⟩ [ Options ]
```

Options:

- -li, -loopInv: enables loop invariant usage

Listing 2: Class Accountant and class Employee

```
public class Accountant {
  public int sum,flag;
  private Employee a, b;
  private int password;
  /*!
  sum flag | a.salary b.salary ;
  flag | password ;
  !* /
  /*@requires true;@*/
  public void magic(){
    if(password>0){
      flag = 1;
      sum = a.salary + b.salary;
    }else{
      flag = 0;
      sum = 0;
    }
  }
}

public class Employee {
  public int salary;
}
```

Listing 3: Declassification specification

```
public class Accountant {
  public int sum,flag;
  private Employee a, b;
  private int password;
  /*!  sum flag | a.salary b.salary ;
  flag | password ; !*
  /*@requires true;
  @escapes  (a.salary + b.salary) \if pass > -1 \to sum ; /
  @*/
  public void magic(){
    ...
  }
}
```

Listing 4: Method calls inside declassification specification

```
public class Accountant {
public int sum,flag;
private Employee a, b;
private int password;
/*!  sum flag | a.salary b.salary ;
flag | password ; !*
/*@requires true;
@escapes  sum(a.salary, b.salary) \if pass > -1 \to sum ; /
@*/
public void magic(){
...
}

/*@ensures \result == x + y ; / @*/
private int sum(int x, int y){
  return x+y;
}

}
```

- -mc, -methodCt: enables loop invariant usage
- -a, -about: shows version and copyright information
- -h, -help: displays usage help
- -d, -depth `number of depth value`: configure the value of number of bounded depth level using in investigating heap structure (default value is 100)

It is convenient to run KEG by adding environment variable PATH the path of KEG's folder. For example, in case PATH has been configured, if we want to check file Factory.java storing in folder `C:\project\accountant` using loop invariant to handle loops and method contract to handle method calls, then the command to check it by KEG is:

```
java -jar exploitgen.jar C:\project\accountant\Factory.java -li -mc
```

References

[1] Beckert, B., Hähnle, R., and Schmitt, P. H. Verification of Object-oriented Software: The KeY Approach. Springer-Verlag, Berlin, Heidelberg, 2007.

[2] Leavens, G. T., Baker, A. L., and Ruby, C. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. ACM SIGSOFT Software Engineering Notes 31, 3 (2006), 1–38.