

## PAPER

# A Static Bug Detector for Uninitialized Field References in Java Programs\*

Sunae SEO<sup>†a)</sup>, Member, Youil KIM<sup>†</sup>, Hyun-Goo KANG<sup>†</sup>, and Taisook HAN<sup>†</sup>, Nonmembers

**SUMMARY** Correctness of Java programs is important because they are executed in distributed computing environments. The object initialization scheme in the Java programming language is complicated, and this complexity may lead to undesirable semantic bugs. Various tools have been developed for detecting program patterns that might cause errors during program execution. However, current tools cannot identify code patterns in which an uninitialized field is accessed when an object is initialized. We refer to such erroneous patterns as *uninitialized field references*. In this paper, we propose a static pattern detection algorithm for identifying uninitialized field references. We design a sound analysis for this problem and implement an analyzer using the Soot framework. In addition, we apply our algorithm to some real Java applications. From the experiments, we identify 12 suspicious field references in the applications, and among those we find two suspected errors by manual inspection.

**key words:** Java object initialization, software verification, program analysis, safety

## 1. Introduction

The Java programming language has powerful features such as inheritance, instantiation and dynamic method binding that enable programmers to implement large and complex Java programs effectively. However, these features lead to complicated program semantics causing programmers to make mistakes and build faulty programs. In recent years, many tools [4], [5], [9], [10], [13] for detecting bug patterns in Java programs have been proposed and proven successful. However, there are still erroneous patterns that those tools do not support.

The uninitialized field reference problem involves the accessing of a field before it is initialized. This bug pattern appears when a Java program tries to instantiate an object which is implemented with class inheritance and dynamic method binding. The mixing of these features gives rise to nontrivial program semantics resulting in faulty code. Moreover, programmers have great difficulty in identifying this bug pattern due to the complexity of the problem.

For example, consider the following case:

```
class Printer {
    Port port = create_port();
    Port create_port() {
        return new NormalPort();
    }
}
class HighSpeedPrinter extends Printer {
    int port_number = 1;
    Port create_port() {
        return new HighSpeedPort(port_number);
    }
}
```

In this example, the designer of `Printer` class intended that the field `port` would be created by overridable method `create_port`; the method creates an instance-specific port. The designer of `HighSpeedPrinter` intended to use `HighSpeedPort` as the instance-specific port and to initialize the port number as 1. At runtime, however, the port number is set to 0 because of the following Java object initialization order:

1. Fields are set to default values<sup>\*\*</sup>; `port_number` is set to 0.
2. `HighSpeedPrinter`'s constructor is invoked.
3. `Printer`'s constructor is invoked.
4. `Printer`'s field is initialized.
5. `Printer`'s constructor is executed.
6. `HighSpeedPrinter`'s field is initialized; `port_number` is set to 1.
7. `HighSpeedPrinter`'s constructor is executed.

This behavior of the Java programs is related to the execution of constructors. In Fig. 1, the class `S` has a constructor whose body is a single print statement. Although the constructor consists of only one line of code, it executes three statements at runtime: it first calls the parent class's constructor, then initializes its own fields, and finally executes its body code. The fields are initialized after the parent constructor call, and invalid access to the fields may occur if the parent constructor reads the fields. In Fig. 1, the field `b` is accessed in the parent constructor `P()` through the function `get_b()`, and is then initialized in the owner class `S`. Clearly, this code pattern shows a *read before initialize* error.

In [2], a warning is given regarding this code pattern along with a recommendation that documentation should be

<sup>\*\*</sup>Note that we look upon this phase as *preparation*, and hence do not regard it as initialization in the present work.

Manuscript received December 15, 2006.

Manuscript revised March 29, 2007.

<sup>†</sup>The authors are with the Division of Computer Science, Korea Advanced Institute of Science and Technology, Korea.

\*This research was supported by the MIC (Ministry of Information and Communication), Korea, under the ITRC (Information Technology Research Center) support program supervised by the IITA. (Institute of Information Technology Assessment) (IITA-2005-C1090-0502-0031)

a) E-mail: sunae.seo@gmail.com

DOI: 10.1093/ietisy/e90-d.10.1663

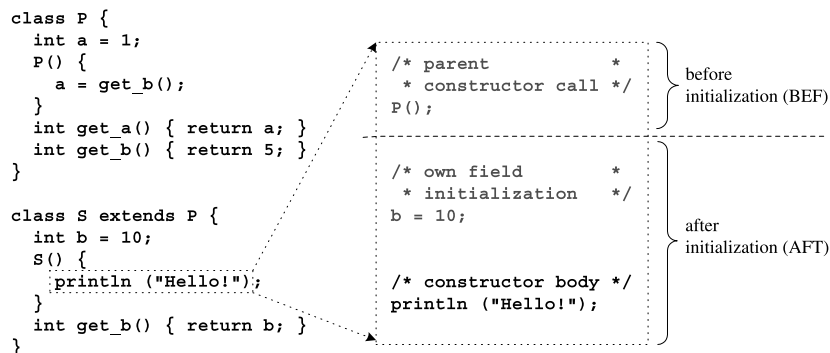


Fig. 1 Execution of constructor.

produced to alert programmers to this problem. Since this code pattern arises through class inheritance, programmers have great difficulty in finding it unless they examine all of the code in the class inheritance chain including the Java library. Moreover, this code pattern cannot be found by existing syntax-based checkers, and requires more sophisticated checkers that understand program semantics. Various Java analysis tools [4], [5], [9], [10], [13] have been developed, and applied to a wide variety of Java programs. However, no existing tool targets the uninitialized field reference problem and provides a method to detect the offending code pattern.

In this paper, we design an effective bug pattern detector for the uninitialized field references. Our detector requires the field usage information in all constructors of classes. This information is computed by our modified set-based analysis [8]. Since the analysis of field usage is path-, flow-, and context-insensitive, our detector based on the analyzer is easy to implement and lightweight to execute. For a clear explanation of our approach, we suggest a simple version of the Java programming language. We explain the design of the field usage analysis using this simplified language and establish the correctness of the analysis along with the formal semantics of the language. In addition, we implement the detector as well as the field usage analyzer in the Soot framework [1], and apply our detector to a selection of widely used Java applications.

The remainder of this paper is organized as follows: In Sect. 2, we demonstrate the detection method for finding the uninitialized field references. In Sect. 3, we introduce the simplified Java language, and present the field usage analysis that computes the field read or write effects in all constructors. Our prototype implementation and the experimental results are described in Sect. 4. In Sect. 5, we discuss related works and conclude our work in Sect. 6.

## 2. Our Approach

The object initialization process becomes quite complex when the class extends another class, and the constructor of the parent class calls an abstract method whose implementation is given in the child class. If the method contains code that accesses the fields of the child class, then illegal access

to the fields occurs as a result of the complex object initialization process. This process is not clearly understood when the parent class is in a library, or when the parent class is not a direct parent of the current class but an ancestor of it.

At the center of this problem, lies the phenomenon of *reading fields before initializing them*. The constructor in a Java program, even one that contains no code, performs two more tasks, super call and then field initialization, as in Fig. 1. When a constructor is called, the super call may access a field that is going to be initialized after the super call.

To handle this problem, we first analyze the constructors and estimate the set of field names that are read or written. The analysis computes two separate sets of field names, one corresponding to the code before initialization and the other to the code after initialization. Given the two sets of field names, we investigate which fields are possibly read before initialization. In this phase, we assume a criterion that only the owner class has the right to initialize its field. Suppose we are given the field names that are written in code after initialization (AFT part in Fig. 1). Among those field names, only some of them are owned by the owner class. For example, the constructor S() in Fig. 2(c) writes another class's field C.c. Although the field C.c is currently written in the AFT part, in accordance with the criterion above, we do not regard it as an initialized field. However, the field P.a written in the constructor S() in Fig. 2(a) is initialized in S(), because P.a is owned by the class S.

Given a class S and two functions *fieldread* and *fieldwritten* that collect the field names read or written from the set of field effects, we define initialized fields in the class S to be:

$$\begin{aligned} & \text{fieldinit}(S) \\ & \triangleq \text{fieldwritten}(\langle \text{AFT} \rangle_S) \cap \text{fieldsOfClass}(S), \end{aligned}$$

where  $\langle \text{AFT} \rangle_S$  is the set of field read and write effects that are collected from the code in AFT part of S's constructor. Then, the fields read before initialization in the constructor of a class S are defined as follows:

$$\text{fieldread}(\langle \text{BEF} \rangle_S) \cap \text{fieldinit}(S). \quad (1)$$

For all classes, we can check whether the set computed from

<pre>class P {   int a, a';   P() { a' = get_a(); }   abstract int get_a(); }  class S extends P{   S() { a = 10; }   int get_a() { return a; } }</pre> <p style="text-align: center;">(a)</p>	<pre>class P {   int a, a';   P() { a = 1; a' = get_a(); }   int get_a() { return a; } }  class S extends P{   S() { a = 10; } }</pre> <p style="text-align: center;">(b)</p>	<pre>class P {   int a;   P() { a = C.get_c(); }   int get_a() { return a; } }  class S extends P{   S() { C.put_c(10); } }</pre> <p style="text-align: center;">(c)</p>
--	---	--

**Fig. 2** Various code patterns for uninitialized field references.

Formula (1) is empty. If the result set is empty, then there is no error pattern, if, however, the set is not empty, there may be a field that is accessed before the owner class initializes it. Among the examples in Fig. 2, this detector diagnoses that examples (a) and (b) may have a field that is read before initialization. In fact, example (a) is an error case, but (b) is not. Code pattern (b) is related to the reinitialization of fields that are inherited from ancestor classes. Such field reinitialization appears frequently in Java programs, and the detector gives a sound but inaccurate answer for all such programs.

To reduce the false alarms caused by field reinitialization, we define the following predicate as our detector for the uninitialized field reference problem: for all classes  $C$ ,

**Predicate 1**  $fieldOfAncestor \cap fieldinit(C) = \emptyset$ , where  $fieldOfAncestor$  is given below:

$$fieldOfAncestor \triangleq (fieldread(\langle BEF \rangle_C) - \bigcup_{c \in ancestors(C)} fieldinit(c)).$$

$fieldOfAncestor$  indicates that the fields initialized by the ancestor classes can be safely accessed all of the time. Naturally, erroneous cases may arise in which an ancestor class accesses a certain field and then initializes it. However, this pattern will be detected by our detector when the detector analyzes the ancestor class. Hence, we assume that the ancestor classes access their fields in a safe way.

For our detector, we need to design an analyzer that computes the field read and write effects for each constructor of a Java program. In the case of a method call, we need more approximation of statically computing the dynamic behavior of method binding. For this approximation, we assume that we already have type information, and that the types including class structures are proved correct. After the analyzer computes the field effects, we check whether Predicate 1 is satisfied for all of the constructors of the program. If it holds, the program has no uninitialized field references; whereas if it does not hold, the program may read a field before initializing it.

### 3. Field Usage Analysis

In this section, we introduce the analysis to compute the field read effects and write effects for each class construc-

$\varphi ::= defn^*$	program
$defn ::= class\ c\ extends\ c\ \{field^*\ constr\ meth^*\}$	class definition
$field ::= var\ fd$	field
$constr ::= cstr()\ \{super();\ cmd\}$	constructor
$meth ::= md()\ \{cmd;\ e\}$	method definition
$e ::= new\ c.cstr()$	create object
$this$	self reference
$null$	null object
$e : c.fd$	read field
$e : c.md()\$	method call
$cmd ::= skip$	skip
$e : c.fd = e$	write field
$cmd; cmd$	sequence of commands
$c ::= a\ class\ name\   Object$	class name
$fd ::= a\ field\ name$	field name
$md ::= a\ method\ name$	method name

**Fig. 3** CoreJava syntax.

tors. To formally address the analysis, we define a simplified version of the Java language. A constraint system for the field usage information is introduced on this simplified language. Finally, using the field usage information given by the constraint system we demonstrate how to check the code pattern of uninitialized field references with the example programs.

#### 3.1 Language: CoreJava

For presentation brevity, we consider only the core constructs of the Java programming language as in Fig. 3. We call it *CoreJava*.

A CoreJava program consists of a sequence of class definitions. Each class defines a set of fields, a constructor and a set of methods. The constructor of a class does basic setup for object creation including field initialization. The program statements consist of commands such as skip, field write and a sequence of commands. A field write command updates a field with the expression's value. The expressions are either an object created, this, null, field read, or a method call. The expressions for class member access are explicitly typed. We assume that those type annotations are inferred by a correct type system. In particular, the field access  $e.fd$  is statically typed so that the type annotated expression  $e : c.fd$  means the field  $fd$  in the class  $c$  even though the type of the expression  $e$  is not  $c$  (if  $e$ 's type is  $c'$ ,  $c$  should be an ancestor

of  $c'$  due to the type correctness.)

Note that CoreJava contains less constructs than the Java language. For instance, the method does not have arguments, the fields are not explicitly type-defined, etc. We simplified those parts out in order to present our idea clearly.

The object creation semantics is composed of two steps: a fresh object is allocated with each field preparation, and the constructor of the object executes over this fresh object. The constructor body is restricted syntactically as Fig. 3. It first calls the constructor of the parent class and then executes its own commands. We assume that the constructor of `Object` is exceptional; `Object` is the top class for all other classes so it does not have the super call command. A Java program that has a field initialization,

```
class A {
  int x = 1;
  A() { // do something.. }
}
```

may be modified in CoreJava as follows:

```
class A extends Object {
  var x
  cstr() { super(); this.A.x = 1;
  // do something.. }
}
```

### 3.2 Constraint Generation System

Our approach to handle the uninitialized field references is to compute the field read and write effects that may occur in a method or a constructor. Each method and constructor consists of commands and expressions, so we analyze the commands and expressions first. The analysis for our problem is designed as set-based analysis [8].

Set-based analysis analyzes programs by approximating the various runtime environments as one set-based environment and estimates the runtime values of program variables as sets of values. For set-based analysis, we first design set equations with set variables that refer to the meaning of each program point, and compute the least solution for the set variables that satisfies the equations.

We modify the traditional set-based analysis to compute the field read and write effects for each program point. An important difference is that our analysis does not use the set-based environment at all; the effects on the fields of classes are not related to the program variables, hence there is no need to maintain the environment. We picked the set-based analysis up because it is intuitive to design and easy to implement.

The set-based analysis consists of *set variables* that keep the set-based information for each program point and *set constraints* that are the equations of set variables, which maintain the set-based information flows. As the notational convention, we use alphabets  $\mathcal{X}, \mathcal{Y}$  for set variables and  $C$  for set constraints. We use subscripts such as  $\mathcal{X}_1, C_1$  for the subexpression or subcommand. If the subscript is 1, the set variable (or set constraint) corresponds to the first subcomponent, if 2, it refers to the second subcomponent and so on.

To distinguish each method syntactically, we use method name with its class containing it such as  $c.md()$ . Sometimes, we use the method name as a subscript to denote the set variable of the specific method. The set variable or the set constraint without any subscript corresponds to the current program component under consideration.

The set constraints that represent the analysis information flows are of the form  $\mathcal{X} \supseteq se$ , where  $se$  is a set expression. A set constraint,  $\mathcal{X} \supseteq se$ , means that the set denoted by  $se$  is contained by  $\mathcal{X}$ . The left-hand side of a set constraint is only one set variable and the right-hand side is a set expression of the following form:

$$\begin{aligned}
 se ::= & \{x \mid x = c.fd_R \vee x = c.fd_W\} \\
 & \mid \mathcal{X} \mid \mathcal{Y} \\
 & \mid seq_c(\mathcal{X}, \mathcal{X}) \\
 & \mid noseq(\mathcal{X}) \\
 & \mid se \cup se
 \end{aligned}$$

$\{x \mid x = c.fd_R \vee x = c.fd_W\}$  is a set of field read and write effects, where  $c.fd_R$  is an effect of reading the field  $c.fd$  and  $c.fd_W$  is of writing the field.  $seq_c(\mathcal{X}, \mathcal{Y})$  means two separate effects in class  $c$ 's constructor, the occurring order of which we want to trace. This set expression is used in the set constraint for the constructors. The subscript  $c$  is the class of the corresponding constructor. The class name  $c$  is used when we collect the potentially initialized fields among written fields. Sometimes we do not have to separate two kinds of effects for the constructors. When the program code contains a constructor call and we are analyzing the program code, we indicate the constructor call part as the set expression  $noseq(\mathcal{X})$ , which means that there is no need to separate two effects contained by  $\mathcal{X}$ .

We design a deduction system for the set constraints of programs in Fig. 4. Given a program, the constraint generation rules generate the set constraints of the program mechanically. In the constraint generation rules, we use four  $\triangleright$  judgments that have the same meaning except the syntactic category indicated by the subscript. The judgment for constructors  $\triangleright_c constr : C$  means that the constructor  $constr$  has the set constraints  $C$ .

The set constraints for a program are generated inductively. The set constraints for a command or an expression are the union of its own set constraints and the set constraints for its subcomponent. For instance, the set constraints for the field read expression,  $e : c.fd$  are generated by **[FdRead]**.

$$\frac{\triangleright_e e : C_1}{\triangleright_e e : c.fd : \{\mathcal{X} \supseteq (\{c.fd_R\} \cup \mathcal{X}_1)\} \cup C_1}$$

The set variable for the field read expression itself contains the effect of reading field  $c.fd$  ( $\mathcal{X} \supseteq \{c.fd_R\}$ ) and all effects included in the set variable for the subexpression  $e$  ( $\mathcal{X} \supseteq \mathcal{X}_1$ ). In addition to the constraint for the expression itself, the set constraints for the field read expression contain the constraints generated from the subexpression  $e$ .

$\frac{\triangleright_c c'.\text{cstr}() : C_{c'.\text{cstr}()} \quad \triangleright_c \text{cmd} : C_2 \quad c' = \text{parent}(c)}{\triangleright_{i,c}.\text{cstr}\{\text{super}(); \text{cmd}\} : \{X \supseteq \text{seq}_c(X_1, X_2), X_1 \supseteq \text{noseq}(X_{c'.\text{cstr}()})\} \cup C_{c'.\text{cstr}()} \cup C_2}$	[Cstr]
$\frac{\triangleright_c \text{cmd} : C_1 \quad \triangleright_e e : C_2}{\triangleright_{m,c}.\text{md}\{\text{cmd}; e\} : \{X \supseteq X_1 \cup X_2\} \cup C_1 \cup C_2}$	[Meth]
$\frac{}{\triangleright_e \text{new } c.\text{cstr}() : \{X \supseteq \text{noseq}(X_{c.\text{cstr}()})\}}$	[New]
$\triangleright_e \text{this} : \emptyset \quad \triangleright_e \text{null} : \emptyset$	[This], [Null]
$\frac{\triangleright_e e : C_1}{\triangleright_e e : c.\text{fd} : \{X \supseteq (\{c.\text{fd}_R\} \cup X_1)\} \cup C_1}$	[FdRead]
$\frac{\triangleright_e e : C_1}{\triangleright_e e : c.\text{md}() : \{X \supseteq (X_{c'.\text{md}()} \cup X_1) \mid c' \in \text{desc}(c)\} \cup C_1}$	[MethCall]
$\triangleright_c \text{skip} : \emptyset$	[Skip]
$\frac{\triangleright_e e_1 : C_1 \quad \triangleright_e e_2 : C_2}{\triangleright_e e_1 : c.\text{fd} = e_2 : \{X \supseteq (\{c.\text{fd}_W\} \cup X_1 \cup X_2)\} \cup C_1 \cup C_2}$	[FdWrite]
$\frac{\triangleright_c \text{cmd}_1 : C_1 \quad \triangleright_c \text{cmd}_2 : C_2}{\triangleright_c \text{cmd}_1; \text{cmd}_2 : \{X \supseteq (X_1 \cup X_2)\} \cup C_1 \cup C_2}$	[Seq]

Fig. 4 Constraint generation rules.

We selectively explain two constraint generation rules out of Fig. 4, a constructor declaration and a method call case. The constraint generation rule for a constructor declaration is [Cstr]:

$$\frac{\triangleright_c c'.\text{cstr}() : C_{c'.\text{cstr}()} \quad \triangleright_c \text{cmd} : C_2 \quad c' = \text{parent}(c)}{\triangleright_{i,c}.\text{cstr}\{\text{super}(); \text{cmd}\} : \left\{ \begin{array}{l} X \supseteq \text{seq}_c(X_1, X_2), \\ X_1 \supseteq \text{noseq}(X_{c'.\text{cstr}()}) \end{array} \right\} \cup C_{c'.\text{cstr}()} \cup C_2}$$

By its syntax, the constructor's body consists of a super call and a subcommand. Since we want to discriminate the effects occurring in the super call and the effects in the subcommand, the set variable for the constructor contains the set variables denoting both effects with the class of the constructor ( $X \supseteq \text{seq}(X_1, X_2)$ ). This is deconstructed with *noseq* when the constructor is invoked in a part of code such as the new object creation ([New]) and the super call execution. Since the constructor itself contains a super call execution, the deconstruction is included as a constraint ( $X_1 \supseteq \text{noseq}(X_{c'.\text{cstr}()})$ ). The constraint generated from the rule for method call ([MethCall]) is not simple as the other rules.

$$\frac{\triangleright_e e : C_1}{\triangleright_e e : c.\text{md}() : \{X \supseteq (X_{c'.\text{md}()} \cup X_1) \mid c' \in \text{desc}(c)\} \cup C_1}$$

By the Java semantics, the method call resolution is determined precisely only in runtime. However, we need to analyze which method would be invoked statically. Hence, we adopt the class hierarchy tree to handle this problem. Given a class hierarchy, the function *desc* gets a class *c* and returns all the descendant classes of *c* in the class hierarchy. For instance, suppose that a class B extends A and that both A and

B have the method `get_class_name()`. Then, when there is a method call `a:A.get_class_name()` in a program, our constraint generation rule would produce the following two constraints according to [MethCall]:

$$\{X \supseteq X_{A.\text{get\_class\_name}()}, X \supseteq X_{B.\text{get\_class\_name}()}\}$$

Note that the subscript of the form *c.md()* means that the corresponding set variable is for the body of the method in the subscript.

In most of cases, the solution of the set constraints is a set for each set variable, which contains the field access effects. In case of constructors, the set variable concerns two sets, each of which corresponds to the effects before super call and after the super call. This kind of set expressions that directly denote the set value for each set variable is called *atomic expression* defined as follows:

$$ae ::= \{x \mid x = c.\text{fd}_R \vee x = c.\text{fd}_W\} \mid \text{seq}_c(X, X)$$

In constraint solving phase, we reduce the constraints in the form of  $X \supseteq ae$ .

In solving phase, some additional rules are applied to the constraints generated by the rules in Fig. 4. One of the additional rules generates the constraints of the form  $X \supseteq ae$  by applying  $\supseteq$  transitively. Two rules deconstruct the ordered effects of the form  $\text{seq}_c(X, Y)$  when we do not have to separate those effects. These addition rules are formally given in Fig. 5.

The set constraint generation produces a finite number of set constraints, and the solution of these set constraints exists all the time; the right-hand side of the set constraints is defined non-decreasingly and the largest set value for each set variable is bound by the number of class fields, which is

finite. Therefore, our set-based analysis to compute the least solution satisfying the set constraints always terminates.

### 3.3 Checking the Field Usage Result

Given the result of the field usage analysis, we investigate the values of the set variables for all constructors. Given the set constraints  $C$ ,  $lm(C)$  is a least model of the constraints, and  $lm(C)(X)$  is the set value for the set variable  $X$  determined by the least model  $lm(C)$ . Two projection functions *fieldread* and *fieldwritten* extract the fields with read effect or with write effect respectively from the set of all effects. Among fields with write effect, we can collect the potentially initialized fields using the subscript class name  $c$  in the equation  $X \supseteq seq_c(X_1, X_2)$ . The final step of our analysis is to check whether Predicate 1 holds or not. If it holds, the program has no uninitialized field references, but if not, the program may read a field before initializing it.

Consider the analysis result of the example in Fig. 1. Though our CoreJava has restricted syntax, we can easily extend the syntax and the analysis. Some of results from the analysis are as follows:

$$\begin{aligned} X_{P.get\_a()} &\supseteq \{P.a_R\} \\ X_{P.get\_b()} &\supseteq \{\} \\ X_{P()} &\supseteq seq_P(\emptyset, \{P.a_W, S.b_R\}) \\ \\ X_{S.get\_b()} &\supseteq \{S.b_R\} \\ X_{S()} &\supseteq seq_S(\{P.a_W, S.b_R\}, \{S.b_W\}) \\ \\ fieldsOfClass(P) &= \{P.a\} \\ fieldsOfClass(S) &= \{P.a, S.b\} \end{aligned}$$

The first effect set for  $X_{P()}$  is empty set because the class **A** has a parent **Object** and we assume that the class **Object** contains nothing. In the second effect set for  $X_{P()}$ , the effect  $P.a_W$  is added by the initialization `int a = 1`. As we noted previously, though the initializations do not appear in the body of the constructor, by the Java semantics those are executed in the constructor just after the super call. The second effect set  $\{S.b_W\}$  for  $X_{S()}$  is produced by the same reason. The constraint for  $X_{P()}$  satisfies the predicate (1), but the case for  $X_{S()}$  does not:

$$(\{S.b\} - \{P.a\}) \cap \{S.b\} = \{S.b\} \neq \emptyset$$

$\frac{X \supseteq Y \quad Y \supseteq ae}{X \supseteq ae}$	
$\frac{X \supseteq noseq(Y) \quad Y \supseteq seq_c(X_1, X_2)}{X \supseteq X_1}$	
$\frac{X \supseteq noseq(Y) \quad Y \supseteq seq_c(X_1, X_2)}{X \supseteq X_2}$	
$\frac{X \supseteq (X_1 \cup X_2)}{X \supseteq X_1}$	$\frac{X \supseteq (X_1 \cup X_2)}{X \supseteq X_2}$

Fig. 5 Constraint solving rules.

Hence, we conclude that the field  $S.b$  may be read before initialized when an object of the class **S** is instantiated. In the same way, the bug pattern in (a) of Fig. 2 is detected and (b) and (c) are proven safe:

- (a)  $(\{P.a\} - \{P.a'\}) \cap \{P.a\} \neq \emptyset$
- (b)  $(\{P.a\} - \{P.a, P.a'\}) \cap \{P.a\} = \emptyset$
- (c)  $(\{C.c\} - \{P.a\}) \cap \{\} = \emptyset$ .

### 3.4 Correctness of Field Usage Analysis

We give the formal semantics of CoreJava language and present the correctness of the field usage analysis. In brief, we define the semantics of CoreJava programs in a store-based big step operational semantics. Then, we modify the semantics and obtain an augmented semantics with the effects on fields of classes. We prove the correctness of our analysis by showing that our analysis is the sound abstraction of the augmented semantics. The detailed abstraction process and the proof are demonstrated in [14].

#### Theorem 1 (Correctness)

*The effects on field read and write analyzed by our set-based analysis are the correct approximation of the real field reads and writes occurring in Java programs.*

## 4. Experimental Results

We have implemented our analysis using Soot [1] which is a framework for optimizing or transforming Java bytecode. The Soot framework provides some preprocessing analyzers such as points-to analysis and call-graph construction. Our prototype analyzer have been implemented as a Soot transformation phase using the Soot API. Our implementation fully depends on the Soot framework, and differs slightly from the designed analysis. Nevertheless, we have proved that our analyzer is a precise and effective detector for the uninitialized field reference problem. The experiments were performed on a PC with two 3.2 GHz XEON processors and 4 GB of memory. We ran our analyzer using Java HotSpot VM build 1.4.2\_03 with a Java heap size of 2 GB for all test cases.

We applied our analyzer to six real Java applications that each contain at least 20,000 lines of source code. Table 1 summarizes the characteristics of the programs. In the column labeled (A) we list the number of constructors; and in the column labeled (B) we list the number of constructors

Table 1 Characteristics of the test cases.

Input	LOC	Classes	(A)	(B)
findbugs-0.7.3	24,582	2272	3136	105
pmd-3.8	39,809	2222	3034	109
jedit-4.2	59,803	4147	5449	382
jext-5.0	60,194	4270	5666	387
jfreechart-1.0.2	101,708	3576	4778	299
soot-2.2.3	116,456	3968	4854	285

**Table 2** Analysis results of the test cases.

Input	Total time	Analysis time	Suppressed Alarm(s)		Alarm(s)	Suspected Error(s)
			Suppressed(1)	Suppressed(2)		
findbugs-0.7.3	135 s	59 s	0	0	1	0
pmd-3.8	156 s	73 s	1	0	1	0
jedit-4.2	316 s	163 s	17	2	1	0
jext-5.0	396 s	196 s	38	42	4	1
jfreechart-1.0.2	235 s	163 s	0	9	1	0
soot-2.2.3	681 s	161 s	2	0	4	1

invoking a method which can be overridden by subclasses; such constructors can be sources of the type of bugs considered in this paper.

The experimental results show that the Java programs all use the uninitialized field reference pattern. Table 2 lists the analyzing time, the number of suppressed alarms, the number of final alarms (except suppressed alarms) and the number of suspected errors found during the analyses. Here, the analyzing time is the entire Soot runtime, including call graph construction, points-to analysis and our analysis. The third column lists the runtime of our fixpoint computation. The analysis examines jdk library code as well as the application code, which is one of the reasons for the long analysis time.

In the column Suppressed Alarms, we demonstrate the number of alarms filtered out by Predicate 1 in Sect. 2 and another effective criterion which excludes the cases that the considered field does not belong to either the parent or the child class but belongs to one of ancestors in the class hierarchy. The alarms in column Suppressed(1) are false alarms caused by field reinitialization, as described in Sect. 2. These alarms can be suppressed by applying Predicate 1. The alarms in column Suppressed(2) are false alarms caused by Java class hierarchy. For example, let us assume that there is a class `C` that inherits `java.awt.Component` of the Java library, and the constructor of `C` invokes some methods that use instances of `java.awt.Component` or descendants of `java.awt.Component`. Then, there might be some access to the fields of `java.awt.Component`. In this case, the field access is not an uninitialized field reference because the fields accessed are not the fields of `C`, but rather the fields of the other objects. However, since `C` also inherits `java.awt.Component`, the analyzer blindly identified them as alarms. This kind of alarm occurs when a program intensively uses the Java library. In our case, the programs that produce alarms in Suppressed(2) have graphical user interfaces, which use numerous Java GUI classes.

The sixth column, Alarm(s), lists the number of uninitialized field reference patterns found during the analyses. These alarms indicate the fields that are read in a parent class before being explicitly initialized by its child class. Interestingly, in most cases, programs avoid faulty execution using conditional statements; for example, before accessing the field, programmers test whether the field is null.

The seventh column, Suspected Error(s), lists the number of serious cases among the alarms in the sixth column. When it is evident that the programmer was already aware of

the uninitialized field reference problem and tried to avoid it via a conditional statement, we consider the case as a mere false alarm. However, when the field is used without an explicit initialization, we consider the case as a suspected error; with the default initialization (a.k.a. *preparation*), such case might not be a real error, but we believe that it is sufficiently dangerous to warrant correction.

Among the alarms found in the analysis, we present one sample code to demonstrate how the problem occurred and how programmers avoided it.

```
// java.util.Random
class Random {
    Random (int v) { init_work(int v); }
    void init_work(int v) { ... }
}

// java.security.SecureRandom
class SecureRandom extends Random {
    SecureRandom () { super(-1); x = 1; }
    void init_work(int v) {
        if (v != -1)
            //reads x
    }
}
```

A designer of the `java.util.Random` class expects that the inherited class implements the `init_work` method. However, one of the inherited classes, `java.security.SecureRandom`, does not allow its `init_work` method to be invoked from `java.util.Random`'s constructor, because it may access the uninitialized field `x`. The designer's decision is to be followed, the programmer who instantiates class `java.security.SecureRandom` should invoke the `init_work` method later to preserve the semantics of `java.util.Random`. In other words, even though the programmer who wrote `java.security.SecureRandom` avoided reading an uninitialized value of `x`, a new hole of semantic bugs is introduced. We conclude that our experiment effectively identified such holes even though we regard them as false alarms.

## 5. Related Works

Previous studies [6] have investigated the object initialization problem arising from access to a premature object,

while here we address the problem of access to a field before its initialization. In the object creation process, the first step is to allocate space for the object, and the second is to initialize the object by executing the user-provided constructor code. After allocation, there are a few subtle situations, such as when the constructor throws an exception, in which an object that has not yet initialized may be used. This object initialization problem is managed in Java bytecode verification [6], [12]. Unlike the object initialization problem, which always arises unintentionally, the references to uninitialized fields may occur on purpose. Hence, a verifier of uninitialized field references is meaningless, but a checker for such references may help programmers to identify and debug some faulty code patterns.

Borger and Schulte [3] pointed out that Java Language Specification [7] underspecifies the the exact moment when the initializer for static fields in classes is executed leading to serious problems such as semantic mismatches in different Java Virtual Machine environments and deadlock of two different classes each waiting for the other's initialization. Kozen and Stillerman [11] provided an algorithm to compute the class initialization dependency, which permits the programmer to check whether his or her code has circular dependency in initializing static classes. While their study focuses on the static fields of classes, we investigated the usage of common (non-static) fields. Computing the class initialization dependency requires analysis of the whole program analysis, whereas references to uninitialized fields can be identified by considering only class structures and type information.

## 6. Conclusion

In this paper, we have proposed a code pattern detector that finds instances of the potentially erroneous usage of fields in object creation processes. In addition, we designed a field usage analysis and provided a formal correctness proof for it. Experiments using the prototype analyzer demonstrated that our detector produces a small number of false alarms and that some Java programs are free of uninitialized field references.

The Java Language Specification [2] recommends that care be exercised in the use of uninitialized field reference patterns. Since such code patterns stem from class inheritance, it is difficult for programmers to trace them out by hand. Using our detection method, however, programmers can identify code patterns in a fully automated fashion, allowing them to avoid error patterns and more clearly understand their code.

In future work, we intend to design a more sophisticated analyzer for the general version of the uninitialized field reference problem. Currently, we have assumed that the initialization position is fixed in each constructor; in general, however, every field has its own initializing position. The location of field initialization is not restricted in the constructor of the owner class but can be in any part of a program. We believe that this analysis will provide impor-

tant solutions to Java security holes.

## References

- [1] Soot: A Java OptimizationFramework. <http://www.sable.mcgill.ca/soot/>
- [2] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Languages*, third ed., Addison-Wesley, 2001.
- [3] E. Borger and W. Schulte, "Initialization problems for java," *Software Concepts and Tools*, vol.19, pp.175-178, 1999.
- [4] J.C. Corbett, M.B. Dwyer, J. Hatcliff, S. Laubach, C.S. Pasareanu, and H. Zheng, "Bandera: Extracting finite-state models from java source code," *Proc. 22nd International Conference on Software Engineering*, pp.439-448, June 2000.
- [5] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata, "Extended static checking for Java," *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp.234-245, June 2002.
- [6] S.N. Freund and J.C. Mitchell, "A type system for object initialization in the java bytecode language," *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, vol.21, pp.1196-1150, Jan. 2000.
- [7] J. Gosling, B. Joy, and G. Steele, *The Java™ Language Specification*, Addison-Wesley, 1996.
- [8] N. Heintze, "Set based analysis of ml programs," *Technical Report, CMU-CS-93-193*, Carnegie Mellon University, July 1993.
- [9] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Notices*, vol.39, pp.92-106, Dec. 2004. <http://findbugs.sourceforge.net>
- [10] JLint, <http://artho.com/jlint>
- [11] D. Kozen and M. Stillerman, "Eager class initialization for Java," *Proc. 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, vol.2469 of *Lecture Notes in Computer Science (LNCS)*, pp.71-80, Springer-Verlag, 2002.
- [12] X. Leroy, "Java bytecode verification: Algorithms and formalizations," *J. Automated Reasoning*, vol.30, pp.235-269, Sept. 2003.
- [13] PMD/Java, <http://pmd.sourceforge.net>
- [14] S. Seo, Y. Kim, H. Kang, and T. Han, "A static bug detector for uninitialized field reference in Java programs," *Technical Memorandum CS-TR-2006-264*, Division of Computer Science, Korea Advanced Institute of Science and Technology, Dec. 2006.



**Sunae Seo** received her B.S. and M.S. degrees in computer science from Korea Advanced Institute of Science and Technology, Korea, in 1998 and 2000, respectively. She is currently a Ph.D. degree student in the Division of Computer Science, Korea Advanced Institute of Science and Technology. Her current research interests include static program verification; especially abstract interpretation, program logic, and programming language theory.



**Youil Kim** received his B.S. and M.S. degrees in computer science from Korea Advanced Institute of Science and Technology, Korea, in 2001 and 2003, respectively. He is currently a Ph.D. degree student in the Division of Computer Science, Korea Advanced Institute of Science and Technology. His current research interests include program verification and optimization using static analysis.



**Hyun-Goo Kang** received B.E. and M.S. degrees in computer science and engineering from Hanyang University, Korea, in 1997 and 1999, respectively. From 1999 to 2000, he was with Electronics and Telecommunications Research Institute, Korea. From 2000 to 2001, he was a research associate at Research On Program Analysis System, Korea Advanced Institute of Science of Technology, Korea. He is currently a Ph.D. degree student in the Division of Computer Science, Korea Advanced Institute of Science and Technology. His current research interests include program analysis and model checking for embedded system.



**Taisook Han** received his B.S. degree in electrical engineering from Seoul National University, Korea in 1976, M.S. degree in computer science from Korea Advanced Institute of Science and Technology, Korea, in 1978, and Ph.D. degree in computer science from University of North Carolina at Chapel Hill, USA, in 1995. He is currently a professor in the Division of Computer Science, Korea Advanced Institute of Science and Technology. His research interests include programming language theory, and design and analysis of embedded systems.