Static Analysis for Computing Escapability and Mutability for Java Components

Aiwu Shi and Gleb Naumovich
Department of Computer and Information Science
Polytechnic University
6 MetroTech Center, Brooklyn, NY 11201, USA
ashi01@cis.poly.edu, gleb@poly.edu

Abstract

A common theme in information security is protection of trusted software components against unauthorized access by untrusted users. In the context of distributed object technologies, such as Enterprise Java Beans, this means preventing leaks of sensitive information to untrusted users, as well as preventing untrusted users from modifying sensitive information.

In this paper, we propose an approach for identification and classification of potentially sensitive information that can leak out of trusted software components to untrusted parties. Unlike the current approaches to securing information flow by extending the type system, our technique is based on static points-to, data- and control-dependence, and object mutability analyses.

1 Introduction

In recent years, distributed object component technologies such as CORBA, COM, and EJB have received wide acceptance. Such technologies enable creation of client-server application that offer developers the convenience of object-oriented development, where server-side functionality is implemented as distributed objects. At run-time, these objects can be accessed by both remote and local clients via remote method calls.

With the explosive growth in the client-server Web applications and reliance on such applications by businesses and general public, security of information handled by distributed object components is becoming increasingly important. The majority of system vulnerabilities are caused not by flaws in communication protocols but rather by exploits of bugs in software applications. Software vulnerabilities can be classified into three general categories:

Information leaks occur when the interface between untrusted clients and a trusted software component returns information that should remain hidden to clients.

Modification access violations occur when the trusted component allows untrusted clients to modify sensitive data controlled by the component.

System resource leaks occur when the trusted component allows untrusted clients to perform sensitive operations on the resources of the underlying computer system, e.g., executing arbitrary commands in a shell.

In this paper, we address the first two vulnerability categories above. We introduce a framework for statically reasoning about potential information leaks from a component by using the concepts from existing static points-to analysis and dependence analysis techniques. In particular, we modify the definitions of data and control dependence to better suit analysis of secure information flow in a software component. Our framework includes algorithms for detecting potential leaks of sensitive information through the boundary of a component.

We address the modification access violation problem by proposing a static analysis that combines points-to and dependence analysis with analysis of mutability of the component state. For the latter, we augment the mutability analysis technique of [26] to increase the analysis precision. To make the discussion concrete, we use the semantics of Java, although in principle other object-oriented programming languages can be handled as well.

This paper makes the following contributions to the field of static analysis for security:

- We propose static analysis of secure information flow based on points-to and dependence analysis, instead of type analysis on which most existing techniques rely.

- We describe novel semantics for dependence analysis, necessitated by the application to secure information flow.

- We introduce an improvement of mutability analysis [26], with application to security of object components.

- We propose a taxonomy of information leaks based on notions of points-to, program dependence, and object mutability.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the new dependence relationships for security analysis and gives definitions as well as illustrations of escapability and mutability. Section 4 gives detailed algorithms for computing escape and mutability information. Section 5 describes a taxonomy of information leaks based on escape and mutability information. Section 6 concludes and discusses future work.

2 Related Work

The related research efforts appear in the areas of secure information flow, mutability analysis, escape and points-to analysis, dependence analysis and alias control and confinement. We survey these areas in the following subsections.

2.1 Secure Information Flow

The flow of information between two entities (e.g., variables) in a program is called secure if both of these entities have the same trust level. Most approaches to ascertaining secure information flow developed to date rely on explicitly or implicitly assigning security levels to program entities.

A recent survey on language-based information-flow security appears in [29]. [3, 4, 32, 34] present approaches to information-flow security based on extending the type system. [1,33] use the semantics-based security model that formalize security in terms of program behavior. The type system and semantics-based security model approaches are combined in [28].

Our proposed algorithm for computing information about values that escape the boundary of a component significantly differs from the existing techniques. Instead of relying on type information, we use points-to graphs for variables of reference types and dependence information for variables of primitive types to compute ways in which information can leak from a component. The main benefit of this technique is that it does not require annotation of types used in the program with respect to their security levels and thus can be used with existing mainstream languages such as Java. Furthermore, since our approach is not specific to a particular points-to or dependence analysis, trade-offs between precision and performance can be exploited for optimal (sufficiently precise but not too expensive) results of the analysis.

2.2 Mutability Analysis

The problem of assuring immutability of objects has been investigated in the past. ConstJava [5], an extension to the Java language, is capable of specifying immutability constraints using the keyword *const* and verifying them at compile time. A framework [24] for immutability specification consists of a set of immutability assertions expressing a richer set of immutability properties, which can benefit code optimization such as load elimination [15], data transformations [13], and global value numbering [27]. Boyland et al. [8] discuss capabilities for sharing and generalize immutability and uniqueness. These capabilities supply an approximation and simplification of many other annotation-based approaches. Unlike the annotation-based approaches, Porat et al. [26] present a static data-flow algorithm to analyze the mutability of fields and classes in Java components.

The mutability analysis described in this paper improves on the mutability analysis of [26]. The improvement is in the area of precision of information computed by this analysis. More specifically, while, similar to [26], our mutability analysis is conservative in the sense that it never claims that a field or class is immutable while in fact it can be modified, there are situations where the analysis [26] determines that a field or class is mutable, while our analysis correctly determines that this field or class is immutable. In addition, unlike [26], we take mutability of *static* fields into account when determining class mutability.

2.3 Escape and Points-to Analysis

Our approach needs information about relationships between references and storage locations in the program, provided by points-to analyses. A large number of points-to analysis techniques have been proposed. A survey of these techniques appears in [17]. Our analysis is not specific to a particular points-to analysis technique and requires only that points-to information computed by this technique is conservative: if a reference can point to a storage location during a real run of the program, the points-to analysis should contain a representation of this relationship. That said, our analysis is sensitive to precision of the points-to analysis it uses.

Pointer escape analysis computes information describing how objects created in one region of a program can be accessed from another region. In the context of our security analysis, we are interested in computing objects created or modified by a trusted component that escape to untrusted clients. A number of escape analysis approaches for object-oriented programs have been proposed [6, 7, 11, 16, 31, 35]. Many of these techniques rely on points-to analysis.

2.4 Dependence Analysis

Dependence analysis computes relationships between program variables that capture the way in which values of these variables depend on one another. A variety of dependence analysis techniques have been proposed. Program dependence graphs (PDG) [14, 25] have been used in approaches for debugging, testing, and maintenance for procedural programs. System dependence graphs (SDG) [18] extend PDGs for sequential procedural programs with multiple procedures. Approaches [10, 20] extend SDGs to apply to object-oriented programming. Zhao [36] extends the SDG specifically for computing dependence information for Java programs.

As described in detail in Section 3.2, the standard dependence information is not sufficient for purposes of our security analysis, since this information does not capture covert channels in the information flow. We extend the standard dependence analysis for object-oriented languages, e.g. [36], to account for covert channels.

2.5 Alias Control and Confinement

The problem of references to security sensitive objects escaping to untrusted parties has been considered in the past. The problem of making sure that no references to a particular object or object type escape a trusted component is referred to in literature as *pointer confinement* [2, 23, 30, 37].

Flexible alias protection [23] is a conceptual model of inter-object relationships which control potential aliasing amongst components of an aggregate object. Confined types and anonymous methods in Java [30] impose a static scoping discipline on dynamic references and loosen confinement to allow code reuse, respectively. Banerjee et al. [2] present a semantic definition of confinement for instance-based alias control, which addresses confinement that is instance-based in that the internal representation (e.g., private fields of an object as Rep) is confined in an object of public class. The instance-based confinement is closely related to the ownership model [12, 21].

Among the approaches above, [2, 12, 21, 37] extend type systems, and [23, 30] are based on mode checking, where programmer supplied alias modes are checked to ensure the protection. In contrast, our approach relies exclusively on relatively low-cost points-to and dependence static analyses.

3 Definitions

In this section, we give definitions of information flow and mutability terms we use throughout this paper.

3.1 Escapability Definitions

The existing escape analyses only compute references that leak outside a program region. In the context of analysis for security, it is important also to track leaks of primitive data outside the component under analysis (CUA). We overload the term *escape* to refer to both references and primitive data values that are in some way available outside the CUA. The recursive definitions in this subsection introduce different ways in which information can escape.

Let set *Objects* represent the set of all run-time objects created by a program on a particular execution. Let set *Fields* represent the set of fields of a given object that are accessible by the object. For a given field f, let *values*(f) be the set containing the possible values of this field[1].

We say that an object (or primitive value) o_2 is reachable from object o_1 if there exists a sequence of field references that starts at o_1 and ends at o_2. Formally, we define set *ReachableFrom* that contains all objects (or primitive values) reachable from a particular object: $\forall o \in$ *Objects*, $p \in$ *ReachableFrom*(o) if

1. $\exists f \in$ *Fields*$(o) : p \in$ *values*(f)
2. $\exists f \in$ *Fields*$(o), \exists r \in$ *values*(f) :
   $p \in$ *ReachableFrom*(r)

For any primitive variable[2] v, let sets *DataDependentOn*(v) and *ControlDependentOn*(v) be the sets of all variables that are data-dependent and control-dependent on v respectively.

Definition: A value of reference or primitive type *escapes* if it directly escapes, reference escapes, partially escapes, data escapes, or control escapes (definitions for these escape types are given below).

Definition: An object *directly escapes* the CUA if a reference to it is made available directly outside the CUA. For example, a `public` method in the CUA may return a reference to this object. Similarly, a variable of primitive type *directly escapes* the CUA if the value of this variable is made available directly outside the CUA.

Definition: An object (or primitive value) o *reference-escapes* the CUA if there exists an object o' that escapes and $o \in$ *ReachableFrom*(o').

Definition: An object o *partially-escapes* if there exists a variable o' (either of reference or primitive type) that escapes and $o' \in$ *ReachableFrom*(o).

[1] Values of a field can be defined either globally or with respect to a particular point in the program. Practically, this depends on the sensitivity of the underlying points-to analysis.

[2] In this paper, variable generally refers to variable as well as field in a Java program.

Definition: A variable v *data-escapes* through variable w if w escapes and $v \in$ *DataDependentOn*(w).

Definition: A variable v *control-escapes* through variable w if w escapes and $v \in$ *ControlDependentOn*(w).

Note that the definitions for reference-, partially-, data-, control-escapes are given with respect to another data that escapes. Such dependencies naturally define sequences of escape information, with each sequence starting at an object or variable that directly escapes. The ideas behind data-escape and control-escape are directly related to the notion of *covert channels* [19].

Consider the code example in Figure 1. According to the definitions above, `int` field `i` data-escapes since `i` is data-dependent on variable `t1` (additional dependence relationship presented in subsection 3.2 is applied) that is returned by method `getData`. `boolean` field `b` control-escapes since `b` is control-dependent on variable `t1` (similarly, additional dependence relationship is applied) that is returned by method `controlData`. Field `v` partially escapes since method `addData` places an object obtained from outside the CUA as parameter `data` to `v` (the object referred to by `data` escapes directly). Information about possible aliases in the component is essential for conservative computation of escape information. For example, consider the code fragment (external to the component in Figure 1 but using this component) below:

```
...; Sample sa=new Sample();
sa.connectBuffer();
StringBuffer sb=sa.exposeBuffer(); ...;
```

Since method `connectBuffer` creates an alias between fields `buffer1` and `buffer2` and method `exposeBuffer` returns `buffer2`, field `buffer1` escapes as well.

3.2 Dependence Relationships for Analyzing Security

There are two types of dependence relationships between primitive variables in a program, *data dependence* representing data flow between variables and *control dependence* representing control conditions on which the execution of a statement or expression depends [9, 14, 25, 36]. Informally, a primitive variable *u* is data dependent on a primitive variable *v* if the value of *v* is used to compute the value of *u*. Similarly, a primitive variable *u* is control dependent on *v* if the value of *v* is used in a conditional statement with a branch on which *u* is assigned a value. It is important to note that both data and control dependence are one-way relationships between variables. For example, analysis of statement $u = v + 1$ determines that u depends on v, but v does not depend on u.

```
public class Sample {
    private Vector v;
    private StringBuffer buffer1;
    private StringBuffer buffer2;
    private StringBuffer buffer3;
    private int i;
    private boolean b;
    public Sample(){
      this.i = 15;
      this.b = true;
      this.buffer1 = new StringBuffer("b1");
      this.buffer3 = new StringBuffer("b3");
    }
    public void connectBuffer(){
      buffer2 = buffer1;
    }
    public String returnString(){
      return buffer3.toString();
    }
    public StringBuffer exposeBuffer(){
      return buffer2;
    }
    public int getData(){
      int t1 = this.i * 2;
      return t1;
    }
    public int controlData(){
      int t1;
      if(this.b)
         t1=0;
      else
         t1=1;
      return t1;
    }
    public void addData(Object data){
      this.v.add(data);
    }
}
```

Figure 1. Code example

In the context of analysis for security, dependence relationships between variables take on a new meaning. Consider again statement $u = v+1$. At the program point where this statement is executed, if the value of u is known outside the CUA, the value of v can be computed as $v = u - 1$. Therefore, for the purposes of our analysis, v data-depends on u in statement $u = v+1$. Similarly, consider a statement $x = y + z$. If values of any two of variables x, y, and z are known to an attacker on the CUA, then the attacker can also figure out the value of the third variable.

Conservatively, control dependence relationship is directly transformed into two-way dependence on each other by our analysis for the purpose of security, i.e., the original dependence relationship that *x* is control dependent on *y* is

TransformDependence()

Input: sets *OldDependenceSet*, *EscapeTuples*
Output: set *DependenceSet*
Steps:

DependenceSet $= \emptyset$
for each tuple $(v_1, v_2, v_3) \in$ *OldDependenceSet*
  if $v_3 =$ CD
    add $(v_1, v_2,$ CD$), (v_2, v_1,$ CD$)$ to *DependenceSet*
  else if v_3 is empty
    add $(v_1, v_2,), (v_2, v_1,)$ to *DependenceSet*
  else if $(v_i, t, v) \cup (v_j, t', v') \in$ *EscapeTuples*,
    where $i, j \in \{1, 2, 3\}, i \neq j$
    add $(v_1, v_2, v_3), (v_2, v_1, v_3), (v_3, v_1, v_2)$
      to *DependenceSet*
  end if
end for

Figure 2. Adding dependence relationships to account for covert channels

extended into the two-way one that both *x* and *y* is control dependent on each other.

Assume that standard dependence information is computed in the form of tuples (v_1, v_2, v_3), representing statement $v_1 =$ *bin_op*(v_2, v_3), where *bin_op* is any binary operation. In case $v_1 =$ *unary_op*(v_2), where *unary_op* is any unary operation, the element v_3 of tuple (v_1, v_2, v_3) is left empty. If the tuple represents control dependence of variable v_1 on variable v_2, the third value in the tuple is flag CD. Let *OldDependenceSet* and *DependenceSet* represent the set of standard dependence information and updated dependence information respectively. Set *EscapeTuples* (see the algorithms in the subsection 4.1 to compute it) represents all of the escape information for the CUA. The algorithm for adding tuples to represent additional dependence information for the purposes of our security analysis is given in Figure 2.

3.3 Mutability Definitions

A component is said to be *immutable* if, once this component is created, it cannot be modified. Porat et al. [26] propose several simple criteria for determining immutability statically: (1) a variable or an object is *immutable* iff its state never changes after the corresponding initialization point; (2) a field is *immutable* iff all the variables that correspond to that field are immutable; (3) a class is *immutable* iff all non-static fields implemented by it are immutable as the state of a class instance is determined by its non-static fields. Similar definitions can be found in [5, 24].

In our work, the mutability definition is modified to add consideration of *static* fields in the analysis of class mutability since static fields also pose important effect on the security analysis and refined to analyze field mutability since fields modifiable outside the CUA are the only focus for the purpose of security. Therefore, precisely, a class is *mutable* iff any one of declared fields including static and non-static ones is mutable and a field is *mutable* iff it can be modified directly or indirectly outside the component.

4 Escape Analysis and Mutability Analysis

4.1 Escape Analysis

We compute escape information for a given component by relying on points-to escape analysis and dependence analysis. Specifically, points-to escape analysis is used to compute information that directly-, reference-, and partially-escapes. Dependence analysis is used to compute information about variables that data- and control-escape.

First, we identify points in the CUA where information is returned outside the CUA. We call such points *return points*. There are four kinds of return points, specific to data that escapes the component:

1. Return statements that return values. The returned value (whether of reference or primitive type) directly escapes.
2. Throw statements. The exception object thrown by the statement directly escapes.
3. For references passed into the component by clients, any point in the component code is a return point, because clients can inspect such references at any time.
4. Similarly, public fields (both static and non-static) of both reference and primitive types can escape at any point.

We compute escape information as a set of tuples of the form (v_1, t, v_2), where

- v_1 is the variable, field, or parameter that escapes the component.
- t identifies the way in which v_1 escapes. The possible values are direct, reference, partial, data, and control, corresponding respectively to directly-escape, reference-escape, partially-escape, data-escape, and control-escape.
- v_2 is a variable through which variable v_1 escapes. For example, tuple $(v_1,$ data$, v_2)$ specifies that v_1 escapes because it data-depends on v_2. One or more other tuples in the set will specify the way(s) in which v_2 escapes. We leave the place-holder for v_2 empty if v_1 directly-escapes.

ComputeEscape()

Input: points-to graphs and dependence information for each point in the code of the CUA
Output: a set of tuples with escape information *EscapeTuples*
Steps:

Initialize the set of escape tuples: *EscapeTuples* $= \emptyset$
Populate *EscapeTuples* with tuples representing directly-escape information.
do
  Invoke *TransformDependence*() to add dependence relationships
  Invoke *ComputeReferenceEscape*() to add tuples describing reference-escape relationships
  Invoke *ComputePartialEscape*() to add tuples describing partially-escape relationships
  Invoke *ComputeDataEscape*() to add tuples describing data-escape relationships
  Invoke *ComputeControlEscape*() to add tuples describing control-escape relationships
while (*EscapeTuples* changed)

ComputeReferenceEscape()

Input: sets *EscapeTuples*, *Fields*(v_1), *values*(f)
Output: updated *EscapeTuples*
Steps:

for all $(v_1, t, v_2) \in$ *EscapeTuples*
  if (v_1 is of reference type) $\wedge (t \neq$ partial)
    for all $f \in$ *Fields*(v_1)
      for all $v \in$ *values*(f)
        add tuple $(v,$ reference$, v_1)$ to *EscapeTuples*
      end for
    end for
  end if
end for

ComputePartialEscape()

Input: sets *EscapeTuples*, *ReachableFrom*(v)
Output: updated *EscapeTuples*
Steps:

for all $(v_1, t, v_2) \in$ *EscapeTuples*
  if (v is of reference type)
    for all v: $v_1 \in ReachableFrom(v)$
      if ($t \neq$ partial)
        add tuple $(v,$ partial$, v_1)$ to *EscapeTuples*
      else add tuple $(v,$ partial$, v_2)$ to *EscapeTuples*
      end if
    end for
  end if
end for

Figure 3. Algorithms for computing escape information for the CUA, part 1

Figures 3 and 4 give our algorithms for computing escape information for a component. The entry point in the algorithms is function *ComputeEscape*() that returns a set *EscapeTuples* of escape information for the CUA. This function repeatedly attempts to add new tuples representing escape information to set *EscapeTuples*, until a fixed point is reached, when set *EscapeTuples* does not change anymore.

ComputeDataEscape()

Input: set *EscapeTuples*, function *ComputeDataDependsOn*(v_1)
Output: updated *EscapeTuples*
Steps:

```
for all (v1, t, v2) ∈ EscapeTuples
    if (v1 is of primitive type)
        for all v ∈ ComputeDataDependsOn(v1)
            add tuple (v, data, v1) to EscapeTuples
        end for
    end if
end for
```

ComputeControlEscape()

Input: set *EscapeTuples*, function *ComputeControlDependsOn*(v_1)
Output: updated *EscapeTuples*
Steps:

```
for all (v1, t, v2) ∈ EscapeTuples
    if (v1 is of primitive type)
        for all v ∈ ComputeControlDependsOn(v1)
            add tuple (v, control, v1) to EscapeTuples
        end for
    end if
end for
```

ComputeDataDependsOn(v)

Input: set *DependenceSet*
Output: set *DataDependentOn* for variable v in the CUA
Steps:

return $\{v_2|(v, v_2, v_3) \in$ *DependenceSet*, $v_3 \neq \mathsf{CD}\}$
$\cup\{v_3|(v, v_2, v_3) \in$ *DependenceSet*, $v_3 \neq empty, v_3 \neq \mathsf{CD}\}$

ComputeControlDependsOn(v)

Input: set *DependenceSet*
Output: set *ControlDependentOn* for variable v in the CUA
Steps:

return $\{v_2|(v, v_2, v_3) \in$ *DependenceSet*, $v_3 = \mathsf{CD}\}$

Figure 4. Algorithms for computing escape information for the CUA, part 2

4.2 Mutability Analysis

In this section, we describe an improvement to the *mutability analysis* of [26]. Our mutability analysis will compute sets *ImmutableClasses* and *MutableClasses* for holding all immutable and mutable classes in the CUA itself and other components used (e.g., standard Java libraries), respectively. *UnknownClasses* is the set of all classes in the CUA for which mutability information is unknown. Set *Methods*(c) represents the set of all declared public methods in a class c. Let *Type*(f) represent the declared type of field f, and *Modifiers*(f) be the set of declared modifiers of field f, including visibility (i.e.,

public, private, default and protected modifiers), final, and static modifiers. For example, if field f is declared as public, static, and final, then *Modifiers*$(f) = \{\texttt{public}, \texttt{static}, \texttt{final}\}$.

We assume that function *ModifiedFields* returns the set of fields in a class modified directly or indirectly by a given public method in this class (e.g., a public method can call a private method in which the field is modified). A conservative algorithm for building this function appears in [22].

Figure 5 contains an algorithm for determining mutability of a given field. This algorithm relies on information about mutability of other classes. Without sufficient class mutability information, a field may remain *unknown*.

Let *Classes* denote the set of all *non-abstract* classes in the CUA, function *AllFields*(c) return the set of all fields in a given class *c*, function *ImmuFields*(c) return the set of fields that have been classified immutable in a given class *c*.

Figure 6 contains an algorithm for determining mutability of classes in the CUA. This algorithm repeatedly calls the algorithm from Figure 5 to refine information about individual fields. Thus, the algorithms in Figures 5 and 6 are interdependent and applied recursively until a fixed point is reached and information about mutability of classes and fields does not change. At this point, all classes that are still not marked as mutable or immutable are conservatively assumed to be mutable.

The algorithms in this section are largely based on those in [26], with two important modifications. While in [26] a field of a mutable type is automatically marked mutable, we only mark such field mutable if it can escape the component and can be modified externally or if public method(s) in the CUA directly or indirectly modify this field. Therefore, our approach increases the precision of mutability analysis. In addition, instead of only considering *non-static* fields for determining class mutability in [26], we take mutability of *static* fields into account for class mutability since any modification of information in the CUA is considered important for security reasons.

For example, refer to code in Figure 1. The approach of [26] will mark field buffer3 mutable because it is of mutable type StringBuffer, defined in the standard Java libraries. However, this field is classified as immutable in our approach since it can not be modified through the declared public methods or other ways after initialization.

DetermineMutable(f)

Input: field f, sets *MutableClasses*, *UnknownClasses*, *EscapeTuples*, *Modifiers*, *Methods*, *values*, *Type*, and functions *ModifiedFields* for each method in the CUA.
Output: classification of mutability of f, one of "mutable", "immutable", "unknown"
Steps:

if $\exists m \in$ *Methods*(c) : $f \in$ *ModifiedFields*(m)
  return "mutable"
else if $\exists$(*values*$(f), t, v_2) \in$ *EscapeTuples*
  if public $\in$ *Modifiers*$(f) \wedge$ final $\notin$ *Modifiers*(f)
    return "mutable"
  else if $\exists$(*values*(f), partial, $v_2)\wedge$
  Type$(v_2) \in$ *MutableClasses*
    return "mutable"
  else if $\exists$(*values*$(f), t, v_2), t \in \{$reference, direct$\}\wedge$
  Type$(f) \in$ *MutableClasses*
    return "mutable"
  else if $\exists(values(f)$, partial, $v_2)\wedge$
  Type$(v_2) \in$ *UnknownClasses*
    return "unknown"
  else if $\exists$(*values*$(f), t, v_2), t \in \{$reference, direct$\}\wedge$
  Type$(f) \in$ *UnknownClasses*
    return "unknown"
  else return "immutable"
  end if
else return "immutable"
end if

Figure 5. An algorithm to determine whether a field is mutable or immutable

We have proposed approaches for statically computing information about escapability and modifiability of objects and variables inside of trusted components. To combine the results of these analyses, Figure 7 presents an algorithm to perform the static analysis for security. This algorithm determines the level of protection of a given field in the CUA against accesses from outside the CUA. Access to a field is classified as safe if the value of the field does not escape the CUA and the field cannot be modified from outside the CUA, visible if the value of the field can escape the CUA but cannot be modified outside the CUA, and modifiable if the value of the field can be modified outside the CUA.

5 Security Analysis

Security policies require protecting the confidentiality and integrity of security sensitive information. However, integrity of mutable objects may be violated by attackers by modifying the objects illegally. Confidentiality of escaped objects may likewise be compromised by attackers able to examine such objects from outside the component.

6 Conclusion and Future Work

This paper proposes a static component security analysis technique that combines points-to, dependence, and mutability analyses for detecting potential sensitive information leaks across the component boundary to untrusted clients, as well as determining the ways in which an untrusted client

DetermineClasses()

Input: set *Classes*, functions *AllFields*(c), *ImmuFields*(c), *DetermineMutable*(f)
Output: set *ImmutableClasses*, *MutableClasses*
steps:

```
Initialize ImmutableClasses to contain all known
immutable classes from libraries used by CUA
Initialize MutableClasses to contain all known mutable
classes from libraries used by CUA
Initialize UnknownClasses = Classes
do
  for all c ∈ UnknownClasses
    for all f ∈ AllFields(c) ∧ f =“unknown”
      result = DetermineMutable(f)
      if result =“mutable”∧ c ∉ MutableClasses
          add c to MutableClasses
          UnknownClasses = UnknownClasses − {c}
      else if result =“immutable”
          add f to ImmuFields(c)
      end if
    end for
    if all f ∈ AllFields(c) ∧ f ∈ ImmuFields(c)
      add c to ImmutableClasses
      UnknownClasses = UnknownClasses − {c}
    end if
  end for
while (UnknownClasses changed)
for all c ∈ UnknownClasses
  add c to MutableClasses
end for
```

Figure 6. An algorithm to determine whether classes are mutable or immutable.

DetermineSecurity(f)

Input: field f, sets *EscapeTuples*, *values*, and functions *ImmuFields*(c) for each $c \in$ *Classes*
Output: classification of field f as safe, visible, or modifiable
Steps:

```
if f ∈ ImmuFields(c),
    where c is the class containing f
    if ∃(values(f), t, v2) ∈ EscapeTuples
        return visible
    else
        return safe
    end if
else
    return modifiable
end if
```

Figure 7. An algorithm to determine whether a field is safe or not

can modify the state of a trusted component. Furthermore, we propose a taxonomy of information leaks based on relationship between fields of a component that represent sensitive data and data that is returned to untrusted clients making calls to methods of the component. While our static technique and taxonomy can be used with any conservative (safe) points-to, dependence, and mutability analysis, this technique is sensitive to precision of these analyses.

We plan to provide the proof of algorithms and implement our technique as a tool with a user interface that allows analysts to specify information in the component that should be considered security-sensitive by the analysis. For example, the analyst may specify that all or only a subset of fields in the component are sensitive.

Once the tool is implemented, we plan to experiment with a variety of Java components, to evaluate its usefulness and efficiency. We will initially concentrate on EJB components, since in many cases they are developed for environments where untrusted clients are granted access to only a subset of functionality and information of a component. The tool will be designed in a way that allows plugging in different points-to, dependence, and mutability analyses. Using different combinations of these analyses, we will evaluate trade-offs between precision of underlying analyses and the rate of false positives (e.g., information leaks that do not present a problem in practice) reported by the tool.

References

[1] J. Agat. Transforming out timing leaks. *Proc. ACM Symp. on Principles of Programming Languages*, pages 40–53, Jan 2000.

[2] A. Banerjee and D. Naumann. A static analysis for instance-based confinement in Java. *Tech. Rep. of Dept. of Comp. and Info. Sci., Kansas State Univ.*, November 2001.

[3] A. Banerjee and D. Naumann. Secure information flow and pointer confinement in a Java-like language. *Proc. IEEE Computer Security Foundations Workshop (CSFW)*, pages 253–267, June 2002.

[4] A. Banerjee and D. Naumann. History-based access control and secure information flow. *Proceedings of the Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart Cards (CASSIS),LNCS Vol. 3362*, pages 27–48, May 2004.

[5] A. Birka. Compiler-enforced immutability for the Java language. *Technical Report MIT-LCS-TR-908, MIT Laboratory for Computer Science*, June 2003.

[6] B. Blanchet. Escape analysis for Java: Theory and practice. *ACM Transactions on Programming Languages and Systems*, 25(6):713–775, 2003.

[7] J. Bogda and U. Holzle. Removing unnecessary synchronization in Java. *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 34(10):35–46, 1999.

[8] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. *Proc. 15th European Conference on Object-Oriented Programming (ECOOP), LNCS Vol. 2072*, pages 2–27, June 2001.

[9] C. Chambers, I. Pechtchanski, V. Sarkar, M. Serrano, and H. Srinivasan. Dependence analysis for Java. *Workshop on Compilers and Parallel Computing*, pages 35–52, 1999.

[10] J. Chan and W. Yang. A program slicing system for object-oriented programs. *Proceedings of the 1996 International Computer Symposium, Taiwan*, Dec 1996.

[11] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 1–19, November 1999.

[12] D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 33(10):48–64, 1998.

[13] J. Dolby and A. Chien. An automatic object inlining optimization and its evaluation. *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 35(5):345–357, 2000.

[14] J. Ferrante, K. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *IEEE Trans. Programming Languages and Systems,*, 9(3):319–349, 1987.

[15] S. Fink, K. Knobe, and V. Sarkar. Unified analysis of array and object references in strongly typed languages. *Proc. 7th International Static Analysis Symposium, LNCS Vol. 1824*, pages 155–174, June 2000.

[16] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. *Proc. 9th International Conference on Compiler Construction, LNCS Vol. 1781*, pages 82–93, 2000.

[17] M. Hind. Pointer analysis: haven't we solved this problem yet. *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 54–61, June 2001.

[18] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, Jan 1990.

[19] B. W. Lampson. A note on the confinement problem. *Comm. of the ACM*, 16(10):613–615, Oct 1973.

[20] L. Larsen and M. Harrold. Slicing object-oriented software. *Proceeding of the 18th International Conference on Software Engineering*, pages 495–505, Mar 1996.

[21] P. Muller. Modular specification and verification of object-oriented programs. *PhD thesis, available from www.informatik.fernuni-hagen.de/pi5/publications.html*, 2001.

[22] G. Naumovich and P. Centonze. Static analysis of role-based access control in J2EE applications. *ACM SIGSOFT, Software Engineering Notes, Workshop on Testing, Analysis and Verification of Web Services*, 29(5):1–10, September 2004.

[23] J. Noble, J. Potter, and J. Vitek. Flexible alias protection. *Proc. European Conference on Object-Oriented Programming (ECOOP), LNCS, Vol. 1445*, pages 158–185, 1998.

[24] I. Pechtchanski and V. Sarkar. Immutability specification and its applications. *Proc. Joint ACM-ISCOPE Java Grande Conference*, pages 202–211, November 2002.

[25] A. Podgurski and L. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Trans. Software Engineering,*, 16(9):965–979, 1990.

[26] S. Porat, M. Biberstein, L. Koved, and B. Mendelson. Automatic detection of immutable fields in Java. *Proceedings of IBM Centre for Advanced Studies Conference (CASCON)*, page 10, 2000.

[27] B. Rosen, M. Wegman, and F. Zadeck. Global value numbers and redundant computations. *Proc. 15th Annual ACM Symposium on the Principles of Programming Languages*, pages 12–27, January 1988.

[28] A. Sabelfeld and H. Mantel. Static confidentiality enforcement for distributed programs. *Proc. Symposium on Static Analysis, LNCS Vol. 2477*, pages 376–394, Sep 2002.

[29] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), January 2003.

[30] J. Vitek and B. Bokowski. Confined types in Java. *Software Practice and Experience*, 31(6):507–532, 2001.

[31] F. Vivien and M. Rinard. Incrementalized pointer and escape analysis. *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 35–46, 2001.

[32] D. Volpano and G. Smith. A type-based approach to program security. *Proceedings of Theory and Practice of Software Development (TAPSOFT), LNCS Vol. 1214*, pages 607–621, 1997.

[33] D. Volpano and G. Smith. Probabilistic noninterference in a concurrent language. *Proceeding of 11th IEEE Computer Security Foundations Workshop*, pages 34–43, June 1998.

[34] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Computer Security*, 4(3):167–187, 1996.

[35] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 34(10):187–206, 1999.

[36] J. Zhao. Applying program dependence analysis to Java software. *Proc. Workshop on Software Engineering and Database Systems, 1998 International Computer Symposium*, pages 162–169, 1998.

[37] T. Zhao, J. Palsberg, and J. Vitek. Lightweight confinement for featherweight Java. *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 38(11):135–148, November 2003.