Inference of Security-Sensitive Entities in Libraries

Yi Lu, Sora Bae, Padmanabhan Krishnan, Raghavendra K.R.
Oracle Labs
Brisbane, QLD 4000, Australia
Email:{yi.x.lu, paddy.krishnan, raghavendra.kr}@oracle.com

***Abstract*—Programming languages such as Java and C# execute code with different levels of trust in the same process, and rely on an access control model with fine-grained permissions to protect program code. Permissions are checked programmatically, and rely on programmer discipline. This can lead to subtle errors. To enable automatic security analysis about unauthorised access or information flow, it is necessary to reason about security-sensitive entities in libraries that must be protected by appropriate sanitisation/declassification via permission checks. Unfortunately, security-sensitive entities are not clearly identified.**

In this paper, we investigate security-sensitive entities used in Java-like languages, and develop a static program analysis technique to identify them in large codebases by analysing the patterns of permission checks. Although the technique is generic, our focus is on Java where `checkPermission` calls are used to guard potential security-sensitive entities. Our inference analysis uses two parameters called proximity and coverage to reduce false-positive and false-negative reports. The usefulness of the analysis is illustrated by the results obtained while checking the OpenJDK7-b147 for conformance to Java Secure Coding Guidelines that relate to the confidentiality and integrity requirements.

***Keywords*-static analysis; Java security; permissions;**

I. Introduction

Tracking information flow from untrusted sources and preventing it from reaching security-sensitive entities is a well known approach to prevent a variety of attacks including injection such as cross-site scripting and SQL injection. Typically, untrusted information needs to be sanitised appropriately before it can reach security-sensitive operations. Once the information is sanitised it has integrity. Similarly, tracking information generated by security-sensitive operations to untrusted destinations can help prevent the leak of confidential information. Confidential information can be suitably declassified depending on the privilege of the recipient. Techniques to ensure confidentiality and integrity typically focus on identifying the correct sanitisers or declassifiers and ensuring that they are applied on all behavioural paths of a program [1]. In the context of languages like Java[1], a sandbox approach is used. Here the execution of untrusted code is restricted via various access control restrictions. Access control restrictions are used in operating systems such as the Android OS where multiple layers of control have been proposed [2]. The key underlying concept is that of security-sensitive operations that access or generate sensitive resources. These operations need to be protected by the access control mechanisms. Unauthorised entities should not be allowed to perform operations on these resources. Both sanitisation and declassification can be viewed as restriction on the information that can be used or generated by security-sensitive operations.

To develop secure systems, the programmer must be able to identify parts of the program that are trusted or not trusted and then perform suitable access control to prevent undesired behaviours. Systems that address these issues include HiStar [3], [4] and Capsicum [5]. However developing large evolving systems that are secure remains a challenge. This is because developers often find it hard to understand the cause of vulnerabilities [6] and the actual security-sensitive operations are not clear [7]. This is especially true when the codebase is large and has evolved over a long time. While behavioural fingerprint can help identify security-sensitive operations [7] the focus of their work, like [1], is to identify locations where access control restrictions can be placed.

In general, sanitisations and declassifications are specified programmatically and there is no formal specification of the restricted entities or permissible operations. Hence individual developers are solely responsible to enforce the desired, but not explicitly specified, security properties of their code. This makes the automatic detection of errors almost impossible as there is no clear specification of acceptable and unacceptable behaviour.

In this paper we address the issue of inferring security-sensitive operations in legacy code. We focus on the Java security model, where permission checks (i.e., calls to the `checkPermission()` method) can be viewed as both sanitisers and declassifiers. That is, the permission checks ensure that information is accepted from or released to entities that have the appropriate privileges ensuring integrity and confidentiality. Furthermore, we focus on the Java Development Kit (JDK) which is a library and thus not a full application. The problem of identifying acceptable behaviour is even harder for libraries like the JDK that are expected to enforce security for all possible programs (that are unknown and may have different privileges) that use the library. Although we focus on the JDK, the technique is applicable to any codebase that uses explicit checks to restrict access, e.g. .NET. The result of such an analysis is required *before* deciding on the kind of sanitisation required on tainted inputs or declassification required on escaped values. That is, the sanitisation/declassification applied to information from/to different sources/destinations will depend on the information and the source/destination.

As the security entities in the JDK are not marked up, we propose a static analysis technique to infer automatically security-sensitive entities that are protected by permission checks. For codebases like the JDK, which are large and have evolved over many years, the security intent is not captured formally. In general, any security analysis, such as those

Sora Bae is currently affiliated with KAIST, South Korea

[1]Java, JDK and JRE are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

© 2017, Yi Lu. Under license to IEEE.
DOI 10.1109/SPW.2017.26

IEEE computer society

implemented to check conformance to the Java Secure Coding Guidelines (JSCG) [8], first needs to identify restricted entities that should be protected. This identification is impossible without fully understanding the semantics of all the entities and their interactions in the program.

The main contributions of in this paper are as follows.

Let *L* be a library that relies on permissions checks for access control, that act as surrogates for sanitisation and declassification.

1) A static analysis technique that identifies entities in *L* that are security-sensitive along with their associated permissions.
2) Detection of errors in access control in *L* introduced when a library developer, who is assumed to be very competent, misses the required permission checks to guard security-sensitive entities.
3) Various uses of security-sensitive entities identified in *L* to ensure relevant security guidelines are enforced.

This paper is organised as follows. In the next section we summarise some of the background material including the Java security model. In Section III we present our formal definition of security-sensitive methods. In Sections IV and V we describe our main contribution, namely, an algorithm to solve the inference problem and its implementation. The usefulness of our inferencing algorithm is demonstrated using a few examples of security analyses in Section VI. The paper concludes by presenting related work in Section VII and summarising our key contributions in Section VIII.

II. Background

In this section we first present a concise summary of permissions in Java [9] and examples of code with permission checks that illustrate the problem we wish to solve. The full documentation on security in Java is available on-line[2].

A. Permissions in Java

Java uses permission checks as an access control mechanism to restrict access to entities that are not universally accessible but accessible to entities that have certain privileges. This access control is enforced dynamically via the `checkPermission(p)` call to ensure all entities on the call-stack have the requisite permission `p` [10]. This mechanism ensures that the untrusted code cannot indirectly access protected items via the trusted code. Although there is no explicit sanitisation process in the JDK, the `checkPermission()` calls act like sanitisation. That is, the value received from an entity that has permission `p` is deemed to be trusted to the level represented by `p`.

We outline some of the issues with the way permissions are specified in Java. Although concrete permissions in Java are objects, different objects can actually represent the same permission. For instance, if all the arguments passed to the constructors of the different objects are the same, the permissions are identical. For example, permissions `p` and `q` in the following two statements `Permission p = new RuntimePermission(s1),Permission q = new RuntimePermission(s2)` will be identical if strings `s1` and `s2` have the same value.

In general, the arguments to the constructor of the permissions come in various forms. In some cases, the permissions are just constant strings (e.g., `RuntimePermission ("getProtectionDomain")` while in other cases the permissions contain user specified parameters (e.g., the value `file` in `FilePermission(file, "read")`). Hence our specification needs to clearly mark the non-constant parameters. Note that, in general, the parameters could be the result of complex computation. For example, `SocketPermission(host, "connect")` where `host = "[" + address.getHostAddress() + "]"` with the value in `address` also computed via a different method invocation. The actual permission value could also be computed; such as `SecurityPermission ("removeProvider."+name)`. Here, a specific provider is to be removed from a list and hence the permission to remove that specific value is required. Permissions could also use wild-cards. For instance, `FilePermission(file, "*")` indicates that all actions related to `FilePermission` will be checked.

Permissions in Java can also be ordered using the implementation-specific `implies()` method. For example, if permission `p` implies permission `q`, an application that has `p` will pass the `checkPermission(q)` call. From the perspective of guarding security-sensitive entities, a `checkPermission(x)` call can be viewed as protecting a security-sensitive entity that requires permission `x` or any other permission implied by `x`.

In summary, our technique needs to handle all the variants of permissions while inferring the security-sensitive methods.

B. Example

Before we describe the problem more precisely and our proposed solution in detail, we present an example, shown in Figure 1, to illustrate the problem we have outlined. The code has a publicly accessible method `entry` that checks for the permission and then invokes the private method `openRead` which then uses some native code to actually open the file. As the method `openRead` is private, the only public access is via the method `entry`. The method `internal` is private and has unrestricted access to the method `openRead`. The method `close` is called after the file has been used and is invoked via the method `done`.

But an analysis that identifies all methods that follow the method to check the permissions is not sufficiently precise. In the above example, one can infer that the application needs permission `p` to access the native method `close` that is perhaps incorrectly identified as security-sensitive. The method `close` may not be security-sensitive; its invocation just happens to occur after the invocation to the method `openRead`. If one is not careful, a large number of methods could be classified as security sensitive. Using such an over-approximated list to detect security violations could lead to a number of false alarms. Therefore, the technique to identify security-sensitive methods needs to be precise.

[2] http://docs.oracle.com/javase/7/docs/technotes/guides/security/index.html

```
1  public class C {
2
3     public void entry(String name) {
4        Permission p;
5        p = new FilePermission(name, "read");
6        checkPermission(p);
7        FileDescriptor fd = openRead(name);
8        done(fd);
9     }
10
11    private FileDescriptor openRead(String name) {
12       return nativeFileOpen(name, "r");
13    }
14
15    private FileDescriptor internal() {
16       FileDescriptor fd = openRead("/etc/passwd");
17       return fd;
18    }
19
20    private void done(FileDescriptor fd) {
21       checkCloseable(fd);
22       close(fd);
23    }
24
25    private void native close(FileDescriptor fd);
26
27    public void anotherEntry() {
28       Permission q;
29       q = new RuntimePermission("getPassword");
30       checkPermission(q);
31       FileDescriptor pfd = internal();
32
33    }
34
35    public void indirect() {
36       checkOkay();
37       FileDescriptor pfd = internal();
38    }
39
40    private void checkOkay() {
41       Permission p;
42       p = new RuntimePermission("getPassword");
43       checkPermission(p);
44    }
45
46 }
```

Fig. 1. Example Program.

The method `anotherEntry()` shown in Figure 1 can be used to access the method `openRead` by invoking `internal`. However, the permission being checked on this path is different to the permission checked on the path from `entry`. Here we have various options for the inference algorithm.

The first option is to also consider the method `internal` as security sensitive that happens to call the security-sensitive method `openRead`. We also need to have a relationship between the permissions checked in `entry` and `anotherEntry`. If no such relation can be established, these different paths (ultimately leading to `openRead`) can be flagged as potential errors made by the programmer.

The second option is to note that access to the the method `openRead` is the disjunction of the two permissions. However, this is not precise. Different parts of the same application could have different permission. So a part that has permission `q` but tries to access `openRead` via `entry` will fail. Hence, for precision, the execution path needs to be part of the description.

The notion of a path and methods on a path needs clarification. In Figure 1 on line 6, there is a direct call `checkPermission`. However, it is possible to call helper methods (as in the method `indirect`) that eventually call `checkPermission`. So it is not sufficient to examine the lexical scope of the methods and `checkPermission` to determine security-sensitive methods. It is essential to consider the control flow path to determine the permissions that guard each candidate method.

To conclude, the key issues are:

- The checking of permissions is not lexically scoped.
- The `checkPermission` calls could be path sensitive.
- A `checkPermission` call guards access to a specific method, but it also appears to guard another method due to the structure of the code.
- The permission values could be the result of some computation.
- The output of our inferencing algorithm must be sufficiently accurate and precise.

III. Defining Security Sensitive Methods and Fields

In this section we define security-sensitive methods and fields. Intuitively, security-sensitive methods perform privileged system operations. However, there is no formal description of such operations making defining security-sensitive methods challenging. We address this challenge by examining the *usage* of potentially security-sensitive methods and *not* by the semantics of the statements inside that method. We do not examine the semantics because such an examination is a manual process and not feasible for large codebases like the JDK. The usage of potentially security-sensitive methods is derived from invocations by the library on behalf of applications.

A public API method that has calls `checkPermission()` is potentially security-sensitive. Such API methods are obvious and do not need any specific analysis. Examples of such method include `java.awt.Toolkit.getSystemEventQueue()` which checks the permission `java.awt.AWTPermission` with the action `accessEventQueue` and `java.lang.Runtime.load()` which checks the permission `java.lang.RuntimePermission` with the action `loadLibrary` concatenated with the name of the library to be loaded. The full list of such methods can be obtained from the JavaSE documentation[3].

But methods that are not directly accessible from the application and do not have a `checkPermission()` in

[3] http://docs.oracle.com/javase/7/docs/technotes/guides/security/permissions.html

them require analysis. Such cases include private methods and methods in restricted packages that are *consistently* guarded by a permission check before they can be invoked.

Definition 1 gives a more precise definition of *potentially* security-sensitive methods.

Definition 1: A potentially security-sensitive method m requiring a permission p is one that satisfies at least one of the following conditions.

1) m is directly accessible by the application and has a `checkPermission()` call in it
2) m is not accessible directly by the application and there is a `checkPermission()` on permission p (outside the lexical scope of `doPrivileged`) on every path from a public API to m.

Note that we do not consider paths involving `doPrivileged` calls as the permission checks may be trivially satisfied in such privileged contexts. We define security read-sensitive and write-sensitive fields in a similar way, i.e, the stores and loads on the field are guarded by permission checks.

Definition 2: A field f is potentially read-sensitive (write-sensitive) requiring a permission p if the following conditions are satisfied.

- f is not accessible directly by the application and
- there is a `checkPermission` on permission p (outside the lexical scope of `doPrivileged`) on every path from a public API to every statement loading from (storing to) f.

Based on the above definitions Definition 3 formally defines security-sensitive operations.

Definition 3: A security-sensitive operation is defined to be any one of the following:

- an invocation of security-sensitive method, or
- a load of a security read-sensitive field, or
- a store to a security write-sensitive field.

Definitions 1 and 2 are sound in that they will identify all security-sensitive methods and fields. However, we need two pragmatic refinements. As shown in Figure 1 we need to conclude that the method `close` is not security-sensitive but at the same time identify `openRead` as security-sensitive. So any generated list of security-sensitive methods needs manual checking because a direct implementation of Definition 3 will report many false positives. To minimise the manual effort we introduce the first pragmatic refinement based on the notion of *proximity*. The second issue is related to programming errors. The permission checks inserted in the program form the basis of our inference algorithm. If a programmer makes a mistake, we will miss a potential candidate. Thus our technique could have false negatives. To avoid this we introduce the second pragmatic refinement based on the notion of *coverage*. These notions also carry over to the definitions of security (read/write)-sensitive fields. We now formalise the notions of proximity and coverage.

a) Proximity: Before we formally define proximity, we say a method n is in the *shadow* of another method m when all the invocations of n are preceded/succeeded in the control flow graph by an invocation of m. A direct implementation of our definitions may incorrectly deem a method n as security-sensitive, when n happens to be in the *shadow* of another truly security-sensitive method m.

Definition 4: Proximity is defined as the length of the shortest sequence of invocations in the call graph from a `checkPermission` invocation to the candidate security-sensitive method.

Proximity measures the closeness, as measured by the number of nested method invocations, of the security-sensitive operation to the permission that is checked. We fix a threshold k and for each `checkPermission` call, identify candidate methods that are guarded within the proximity factor of k.

b) Coverage: If a programmer was remiss in inserting a `checkPermission` call on some control flow path for a truly security-sensitive method, the method will not be identified as security-sensitive. It will be useful to present such scenarios to the library developers who can then exercise their judgement and remedy the situation by placing appropriate invocations of `checkPermission`. In this regard, we extend Definition 1 with thresholds that we call *coverage*.

Definition 5: Given a directed acyclic control flow graph, we define a 'probabilistic coverage' factor for each node as follows:

- The node that defines the specific potential security-sensitive method is assigned the value 1.
- If a node n has value w and has k predecessors, the value assigned to each of the predecessors of n is w/k.
- The value assigned to a given node n is the sum of the all values derived from its successors.

Thus coverage is a measure of the likelihood of a permission check guarding a security-sensitive operation based on the number of paths to the security sensitive operation. At each branch point, we assume that each branch is equally likely and thus spread the possibility uniformly to each branch.

We use the example in Figure 2 to explain the coverage calculation. Assume that m is the security-sensitive method and that n_1 and n_4 represent the checking of the permission p. Nodes n_1 and n_5 have the value 0.5 while the nodes n_3 and n_4 have values 0.25. Thus the coverage for m with respect to permission p is 0.75. In other words, there is path from pe to m (via n_2, n_3 and n_5) with weight 0.25 that does *not* check permission p.

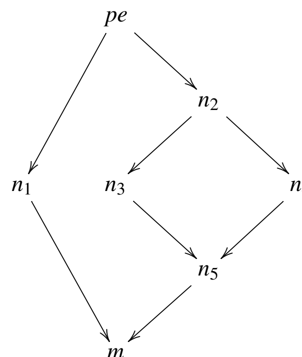

Fig. 2. Coverage Example.

Using Definitions 3, 4 and 5 we define the problem ad-

dressed in this paper as follows. The aim is to design and implement a program analysis technique that can identify security-sensitive entities (such as methods, fields) in libraries that are protected by an explicit access control check. The result of the analysis should also identify a description of the behaviour and the required access control credentials necessary to access the security-sensitive entity. The analysis should also be tunable with user specified proximity and coverage values.

IV. Technique: Algorithm

Algorithm 1 Inference Algorithm.

```
1: Input: program - a library program with codebased access
   control
2: Parameters: proximity P and coverage C thresholds
3: Output: permissions[m, p] - m is security-sensitive method w.r.t.
   permission p that satisfy coverage
4: Error Report: permissions[m, p] - m is security-sensitive method
   w.r.t. permission p that do not satisfy coverage
5: procedure INFERSSM(program)
6:     potentialCandidates = IdentifyPotentialCandidates(program)
7:     for each m in potentialCandidates do
8:         paths[m] = GeneratePaths(m, program)
9:     end for
10:    for each path in paths[m] do
11:        permissions[m, p] = CheckPermissions(path, P)
12:    end for
13:    for each m do
14:        if Coverage(paths[m], permissions[m,p], C) then
15:            Output(permissions[m,p])
16:            if !Coverage(paths[m], permissions[m,p], 1) then
17:                Error-report(permissions[m,p])
18:            end if
19:        end if
20:    end for
21: end procedure
```

Algorithm 2 Identification of Potential Candidates.

```
1: function IDENTIFYPOTENTIALCANDIDATES(program)
2:     potentialList = ∅
3:     for each method m in program do
4:         if m is private or m in restricted package then
5:             potentialList = potentialList ∪ {m}
6:         end if
7:         if m is public and m has a call to checkPermission() then
8:             potentialList = potentialList ∪ {m}
9:         end if
10:    end for
11:    return potentialList
12: end function
```

Algorithm 3 Identify Permissions Checked.

```
1: function CHECKPERMISSIONS(path, P)
2:     permList = ∅
3:     for each invocation i in Proximity(path,i, P) do
4:         if i is checkPermission(perm) then
5:             permList = permList ∪ FullSet(perm)
6:         end if
7:     end for
8:     return permList
9: end function
```

In this section, we describe our technique to infer potential security-sensitive methods. Algorithm 1 presents the key steps in the process. The steps of identifying potential candidates and recording the permissions that are checked along a path are in Algorithms 2 and 3 respectively. As is standard, the method 'GeneratePaths(m, p)' at Line 8 returns all the paths from the public entry to the invocation of the method m in the program p. Note that other auxiliary program analysis techniques, that are standard such as call-graph construction using points-to analysis [11] and data-flow algorithms [12], are used to generate all the paths.

In Algorithm 1 the procedure INFERSSM identifies the security-sensitive methods in the given program. It uses the functions "IdentifyPotentialCandidates" at Line 6 (described in Algorithm 2), "CheckPermissions" at Line 11 (described in Algorithm 3) and "Coverage" at Line 16 that determines if the paths to the method 'm' that have the requisite permission 'p' satisfy the coverage requirement. This uses a standard interprocedural dependence analysis and is not reproduced here. If the coverage requirement is satisfied, 'm' is outputted as a security-sensitive method along with the requisite permission 'p'. Furthermore, we are interested to know which security-sensitive methods are not completely guarded by permission checks on the requisite permission 'p'. When the coverage is not full, i.e., the method 'm' is not always guarded by the permission 'p', an access control error is reported. This list of errors is useful to the library developer to ascertain whether the absence of permission checks were intended or inadvertently missed.

In Algorithm 3 the function 'Proximity' at Line 3 also uses standard graph-theoretic algorithm (such as shortest path) on the control flow paths of the program and is elided. The method 'FullSet' at Line 5 returns all the permissions identified by the specific permission used. This includes permissions that are implied, as explained in Section II-A.

V. Implementation and Initial Experimental Results

The above technique is implemented in the Parfait [13] framework. As we are interested only in the control flow paths when the *security manager* is enabled, our analysis ensures that the code we analyse is guarded by the `if System.getSecurityManager() != null` check. Other auxiliary methods to identify the invocation of `checkPermission()`, to resolve the actual permissions (where possible – otherwise we use only the type of the permission and the part of the action and target that can be resolved) used in permission checks, the call depth from a given point are also defined.

As a case study, we use the OpenJDK7-b147 to study the effectiveness of our approach. We first present examples of security-sensitive methods inferred by our analysis and then present the overall findings. The entire program for these examples can be found on-line[4].

The first example of a security-sensitive method is `removeMBeanServer()` which is **`private`** in the class `javax.management.MBeanServerFactory` and

[4]http://grepcode.com

requires the permission
`javax.management.MBeanServerPermission` with the action `releaseMBeanServer`. This method has been identified because the public method `releaseMBeanServer()` calls the method `checkPermission` (on line 151) before invoking `removeMBeanServer()`.

The second example is the private native method `open()` in `java.util.zip.ZipFile` which requires the `java.io.FilePermission` permission with the `read` action; but the permission also takes the name of the file as a string whose value cannot be determined. The method `open()` is called by the constructor that invokes the method `checkRead()` (on line 205) that has the requisite permission check.

The final example is the private method `loadLibrary0()` in `java.lang.ClassLoader` and is invoked numerous times. But each invocation is guarded by the permission `java.lang.RuntimePermission`. There the action is the concatenation of the string `loadLibrary` and the actual name of the library. This derivation requires the interprocedural analysis to determine the location of the permission check calls.

Table I gives the number of inferred security-sensitive methods in the OpenJDK7-b147 for different coverage and proximity measures. These results show the effectiveness of our approach. Although the classification of reports into True Positives (TP) and False Positives (FP) is done manually, a completely manual process to identify close to 100 methods from the entire codebase consisting of more than 150,000 methods will be tedious. Thus our technique automates a large aspect of the identification process. The results also shows that there are subtle omissions of permissions checks in the JDK. In many of these cases, the omission is deliberate and can only be understood by the developer. For example, the method `sequence()` in `java.lang.Shutdown` which is called by `exit()` and `shutdown()`. While `exit()` performs security checks before invoking `sequence()`, the method `shutdown()` does not because it is invoked by the JVM directly and is assumed to be trusted.

We are able to detect such methods with a coverage of 0.5 but at the cost of increased FPs. Similarly, an increase in proximity results in a significant increase in the number of FPs reports as more methods are under the shadow of security-sensitive invocations.

VI. Use cases

We now outline two potential uses of the security-sensitive inferencing mechanism. These use cases relate to sections from the JSCG [8] which lists several guidelines for JDK library developers. The notion of security-sensitive methods and fields are crucial for analysing the JDK and to determine whether the coding guidelines are satisfied.

In general, JDK developers use the `doPrivileged` API to change the permissions held by the current call stack. Often the permissions are elevated to "all permissions" to perform necessary system operations. However use of security-sensitive entities in a privileged context can lead to security errors caused by improper privilege elevation. Here we give examples of how security-sensitive methods and fields are important in analysing two such coding guidelines.

JSCG 9.3: Integrity in Privileged Contexts: Section 9.3 of JSCG [8] advises JDK developers not to use tainted data inside a `doPrivileged`. This is to prevent an application without the requisite permissions from influencing a security-sensitive operation. Our static analysis achieves this as follows. The first step is to check if a tainted data reaches `doPrivileged`. The second step is to check if that tainted data is then passed on to a security-sensitive method requiring a permission, say p, or stored in a security write-sensitive field requiring a permission, say p. If there is no invocation of `checkPermission` on that p from a public entry to the doPrivileged, then a violation is raised.

Figure 3 gives an example of a violation of JSCG 9.3 along with relevant parts of the `java.lang.System` class. Here `method`, which is publicly accessible, writes to security-sensitive field `props` of `System` class via an invocation of `doPrivileged`. As there is a tainted data `str` inside `doPrivileged` that flows to the security-sensitive operation of writing to security write-sensitive field `props`, we have a violation. Note that to implement an analysis to identify violations of such a guideline we need to first know about security sensitive entities. In this example, we used our technique to identify the field `props` of System class as security-sensitive.

JSCG 9.5: Escape of Sensitive Values: Section 9.5 of JSCG [8] advises JDK developers not to pass the results of the operations in privileged context. Figure 3 also represents a violation of this guideline. Here the value in the security read-sensitive field `props` of `System` class escapes to the application unguardedly (without any invocations of `checkPermission`). As identified `props` field of `System` class is security read-sensitive and is used to detect the violation of the JSCG 9.5 guideline.

VII. Related Work

Much of the research literature in the area of stack inspection has focused on formal security models. Wallach and Felten [10] present a semantics of stack inspection in terms of authentication logic. A Java program may be rewritten into another program that integrates Java stack inspection mechanism in security-passing style. This makes explicit the security environment as an extra argument passed to every function and allows the JVM to use standard optimisation techniques like dead-code and tail-recursion elimination. Fournet and Gordon [14] provides an alternative semantic in λ-calculus with an equational theory, which allows to reason about compiler transformations. Like previous work, they focus on correct compiler transformations—program optimisations like function inlining and tail call elimination which can work correctly in presence of stack inspection. Pistoia, Banerjee and Naumann [15] proposes a new security model, which extends stack inspection with information flow control. It enforces an integrity security property in applications, providing stronger security guarantee than stack inspection alone. These security models do not attempt to define or identify security-sensitive entities.

Pistoia et al. [16] use taint analysis along with `checkPermission()` calls to identify portions of code that

TABLE I
RESULTS ON OPENJDK7-B147.

		Proximity			
		<= 2		<= 3	
		True Positives	False Positives	True Positives	False Positives
Coverage	≥ 0.5	81	28	121	117
	≥ 0.75	64	22	101	86
	$= 1$	63	20	96	77

```
public class LibClass {
    public String method(final String str) {
        return AccessController.doPrivileged(new PrivilegedAction<String>(){
            public String run(){
                String k = System.getProperty("java.class.path");
                System.setProperty("java.class.path", str);
                return k;
            }
        });
    }
}

public final class System {
   ...
   private static Properties props;
   ...
   public static String setProperty(String key, String value) {
      ...
      return (String) props.setProperty(key, value);
   }
}
```

Fig. 3. An Example Illustrating Violation of JSCG 9.3 and JSCG 9.5.

need to be executed in privileged mode to avoid throwing security exceptions. Apart from identifying where privilege elevation is necessary they also identify situations where privilege is elevated unnecessarily. Koved et al. [17] also use `checkPermission()` calls to determine the policy required for the application to function properly. While the core components of the analysis used in both these papers is the same as ours (i.e., finding paths in the call-graph involving `checkPermission()` calls), the aims are quite different. Firstly, we do not consider `doPrivileged()` call nor do we consider the application. The aim is to identify the actual security-sensitive operations which are only implicit in the program but protected by `checkPermission()` calls. Hence we also need to consider shadow of methods and coverage of paths. As we analyse only the library, the required policy and `doPrivileged()` which is required to elevate privileges beyond what is assigned by the policy are not relevant in our analysis. Intuitively, our analysis examines behaviours where a `checkPermission()` precedes other calls while both [17], [16] are interested in calls that ultimately reach a `checkPermission()` call. Security analyses of Java programs will need to combine these different ideas.

In contrast to previous work, our aim is to identify precisely security-sensitive entities in library code and infer the programmer's design intent about how these entities should be protected (i.e., permissions to guard them). The results from our technique may be useful to develop new security models (or refine existing ones) to precisely model security aspects in both applications and libraries.

Wu and Larus [18] present an algorithm that statically estimates program profiles. It estimates branch probabilities with a combination of branch prediction heuristics, and propagates these branch probabilities along each procedure's control-flow graph to obtain local block and edge frequencies. With the function invocation frequencies, it can obtain global block and edge frequencies. Ramalingam [19] extends the dataflow analysis with probability facts to determine not just whether some fact may or may not hold at a program point, but also the probability that the fact may hold true at a given point. The analysis itself does not consider how these probabilities are obtained, but takes them as given by using various heuristics or profile data. Baah et al. [20] present probabilistic program dependence graph to facilitate probabilistic analysis and reasoning about uncertain program behaviours. It augments a program dependence graph with estimates of statistical dependences between node states, which are computed from the test set. Unlike previous work that estimate probabilities with heuristics or test data in whole programs, our technique

reasons about coverage in libraries (partial programs) without knowing any application code and input data. In our analysis, the coverage metric represents the fraction of paths on which a candidate is guarded by a permission check from public access. The coverage is computed based on an interprocedural probabilistic control dependence analysis, starting from a candidate of security-sensitive entities and propagating and distributing probabilities in a backward manner. Hence, it is a purely static approach and does not require any test cases.

AutoISES [21] is perhaps the closest to our work where they infer security specifications (or rules). While they use a notion of security check functions (i.e., methods similar to `checkPermission`) to identify potential targets, their focus is on semantics rules such as the set of data structure accesses that must be protected. Our use case is more direct, we want to identify single methods that represent the security-sensitive operation and not a collection of rules that represent secure behaviour.

VIII. Conclusion

In this paper we have presented a pragmatic approach to identifying security-sensitive entities in the context of the Java permission model. Furthermore, we have focused on libraries (the JDK) where no application is available. The general problem we solve is to identify unknown security-sensitive methods that are protected by know sanitisers or declassifiers. We can detect cases where these known sanitisers and declassifiers are potentially missing. We have implemented our algorithm using standard program analysis techniques [12], [11]. We have presented examples of analyses that use the inferred security-sensitive methods to detect potential security vulnerabilities.

References

[1] B. Livshits and S. Chong, "Towards fully automatic placement of security sanitizers and declassifiers," in *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 2013, pp. 385–398.

[2] M. Backes, S. Bugiel, S. Gerling, and P. von Styp-Rekowsky, "Android security framework: Extensible multi-layered access control on android," in *ACSAC*. ACM, 2014.

[3] N. Zeldovich, S. Boyd-Wickizer, and D. Mazieres, "Securing distributed systems with information flow control," in *Symposium on Networked Systems Design and Implementation*, 2008.

[4] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazieres, "Making information flow explicit in HiStar," *CACM*, pp. 93–101, 2011.

[5] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical capabilities for unix," in *USENIX Security*, 2010.

[6] T. Thomas, B. Chu, H. Lipford, J. Smith, and E. Murphy-Hill, "A study of interactive code annotation for access control vulnerabilities," in *Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2015.

[7] V. Ganapathy, D. King, T. Jaeger, and S. Jha, "Mining security-sensitive operations in legacy code using concept analysis," in *International Conference on Software Engineering*. IEEE Computer Society, 2007, pp. 458–467.

[8] Oracle, "Secure Coding Guidelines for Java SE," http://www.oracle.com/technetwork/java/seccodeguide-139067.html, April 2014.

[9] L. Gong, G. Ellison, and M. Dageforde, *Inside Java 2 Platform Security*, ser. The Java Series. Addison Wesley, 2003.

[10] D. S. Wallach and E. W. Felten, "Understanding java stack inspection," in *IEEE Symposium on Security and Privacy*, 1998, pp. 52–63.

[11] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA)*. ACM, 2009, pp. 243–262.

[12] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*, 2nd ed. Springer, 1999.

[13] C. Cifuentes, N. Keynes, L. Li, N. Hawes, and M. Valdiviezo, "Transitioning Parfait into a development tool," *IEEE Security & Privacy*, vol. 10, no. 3, pp. 16–23, 2012.

[14] C. Fournet and A. D. Gordon, "Stack inspection: theory and variants," in *The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002, pp. 307–318.

[15] M. Pistoia, A. Banerjee, and D. A. Naumann, "Beyond stack inspection: A unified access-control and information-flow security model," in *IEEE Symposium on Security and Privacy*, 2007, pp. 149–163.

[16] M. Pistoia, R. J. Flynn, L. Koved, and V. C. Sreedhar, "Interprocedural analysis for privileged code placement and tainted variable detection," in *ECOOP 2005 - Object-Oriented Programming, 19th European Conference*, 2005, pp. 362–386.

[17] L. Koved, M. Pistoia, and A. Kershenbaum, "Access rights analysis for java," in *Proceedings of the 17th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA)*. ACM, 2002, pp. 359–372.

[18] Y. Wu and J. R. Larus, "Static branch frequency and program profile analysis," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, 1994, pp. 1–11.

[19] G. Ramalingam, "Data flow frequency analysis," in *ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI)*, 1996, pp. 267–277.

[20] G. K. Baah, A. Podgurski, and M. J. Harrold, "The probabilistic program dependence graph and its application to fault diagnosis," in *ACM/SIGSOFT International Symposium on Software Testing and Analysis ISSTA*, 2008, pp. 189–200.

[21] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou, "Autoises: Automatically inferring security specifications and detecting violations," in *Proceedings of the 17th Conference on Security Symposium*. USENIX Association, 2008, pp. 379–394.